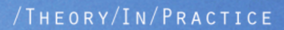


Beautiful Code

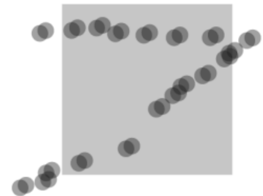
Leading Programmers Explain How They Think

O'REILLY®

Edited by Andy Oram & Greg Wilson

Copyrighted material

Beautiful Code



Edited by Andy Oram and Greg Wilson

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

Copyrighted material

Beautiful Code
Edited by Andy Oram and Greg Wilson

Copyright © 2007 O'Reilly Media, Inc. All rights reserved. Printed in the United States of America.

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

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

Production Editor: Marlowe Shaeffer

Copyeditor: Sanders Kleinfeld

Proofreader: Sohaila Abdulali

Indexer: Ellen Troutman Zaig

Cover Designer: Randy Comer

Interior Designer: Marcia Friedman

Illustrator: Jessamyn Read

Printing History:

June 2007: First Edition.

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

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

This book uses RepKover™, a durable and flexible lay-flat binding.

ISBN-10: 0-596-51004-7
ISBN-13: 978-0-596-51004-6
[C] [8/07]

Copyrighted material

CONTENTS

FOREWORD xv
by Greg Wilson

PREFACE xvii

1 A REGULAR EXPRESSION MATCHER 1
by Brian Kernighan

The Practice of Programming 2
Implementation 3
Discussion 4
Alternatives 5
Building on It 6
Conclusion 8

2 SUBVERSION'S DELTA EDITOR: INTERFACE AS ONTOLOGY 11
by Karl Fogel

Version Control and Tree Transformation 12
Expressing Tree Differences 16
The Delta Editor Interface 17
But Is It Art? 23
Abstraction As a Spectator Sport 25
Conclusions 27

3 THE MOST BEAUTIFUL CODE I NEVER WROTE 29
by Jon Bentley

The Most Beautiful Code I Ever Wrote 30
More and More with Less and Less 31
Perspective 36
What Is Writing? 38
Conclusion 39
Acknowledgments 40

4 FINDING THINGS 41
by Tim Bray

On Time 41
Problem: Weblog Data 42
Problem: Who Fetched What, When? 50
Search in the Large 55
Conclusion 57

Copyrighted material

5 CORRECT, BEAUTIFUL, FAST (IN THAT ORDER): LESSONS FROM DESIGNING XML VERIFIERS 59
by Elliotte Rusty Harold

The Role of XML Validation 59
The Problem 60
Version 1: The Naïve Implementation 62
Version 2: Imitating the BNF Grammar O(N) 63
Version 3: First Optimization O(log N) 64
Version 4: Second Optimization: Don't Check Twice 66
Version 5: Third Optimization O(1) 68
Version 6: Fourth Optimization: Caching 72
The Moral of the Story 74

6 FRAMEWORK FOR INTEGRATED TEST: BEAUTY THROUGH FRAGILITY 75
by Michael Feathers

An Acceptance Testing Framework in Three Classes 76
The Challenge of Framework Design 78
An Open Framework 79
How Simple Can an HTML Parser Be? 80
Conclusion 83

7 BEAUTIFUL TESTS 85
by Alberto Savoia

That Pesky Binary Search 87
Introducing JUnit 89
Nailing Binary Search 90
Conclusion 102

8 ON-THE-FLY CODE GENERATION FOR IMAGE PROCESSING 105
by Charles Petzold

9 TOP DOWN OPERATOR PRECEDENCE 129
by Douglas Crockford

JavaScript 130
Symbol Table 131
Tokens 132
Precedence 133
Expressions 134
Infix Operators 134
Prefix Operators 136
Assignment Operators 137
Constants 138
Scope 138
Statements 140

Copyrighted material

Functions 143
Array and Object Literals 144
Things to Do and Think About 145

10 THE QUEST FOR AN ACCELERATED POPULATION COUNT 147
by Henry S. Warren, Jr.

Basic Methods 148
Divide and Conquer 149
Other Methods 151
Sum and Difference of Population Counts of Two Words 152
Comparing the Population Counts of Two Words 153
Counting the 1-Bits in an Array 154
Applications 158

11 SECURE COMMUNICATION: THE TECHNOLOGY OF FREEDOM 161
by Ashish Gulhati

The Heart of the Start 162
Untangling the Complexity of Secure Messaging 163
Usability Is the Key 165
The Foundation 167
The Test Suite 172
The Functioning Prototype 172
Clean Up, Plug In, Rock On... 173
Hacking in the Himalayas 177
The Invisible Hand Moves 182
Speed Does Matter 184
Communications Privacy for Individual Rights 185
Hacking the Civilization 185

12 GROWING BEAUTIFUL CODE IN BIOPERL 187
by Lincoln Stein

BioPerl and the Bio::Graphics Module 188
The Bio::Graphics Design Process 192
Extending Bio::Graphics 210
Conclusions and Lessons Learned 214

13 THE DESIGN OF THE GENE SORTER 217
by Jim Kent

The User Interface of the Gene Sorter 218
Maintaining a Dialog with the User over the Web 219
A Little Polymorphism Can Go a Long Way 221
Filtering Down to Just the Relevant Genes 224
Theory of Beautiful Code in the Large 225
Conclusion 228

Copyrighted material

14 HOW ELEGANT CODE EVOLVES WITH HARDWARE:
THE CASE OF GAUSSIAN ELIMINATION 229
by Jack Dongarra and Piotr Luszczek

The Effects of Computer Architectures on Matrix Algorithms 230
A Decompositional Approach 232
A Simple Version 233
LINPACK's DGEFA Subroutine 235
LAPACK DGETRF 237
Recursive LU 240
ScaLAPACK PDGETRF 243
Multithreading for Multi-Core Systems 247
A Word About the Error Analysis and Operation Count 250
Future Directions for Research 251
Further Reading 252

15 THE LONG-TERM BENEFITS OF BEAUTIFUL DESIGN 253
by Adam Kolawa

My Idea of Beautiful Code 253
Introducing the CERN Library 254
Outer Beauty 255
Inner Beauty 261
Conclusion 266

16 THE LINUX KERNEL DRIVER MODEL: THE BENEFITS
OF WORKING TOGETHER 267
by Greg Kroah-Hartman

Humble Beginnings 268
Reduced to Even Smaller Bits 273
Scaling Up to Thousands of Devices 276
Small Objects Loosely Joined 277

17 ANOTHER LEVEL OF INDIRECTION 279
by Diomidis Spinellis

From Code to Pointers 280
From Function Arguments to Argument Pointers 282
From Filesystems to Filesystem Layers 285
From Code to a Domain-Specific Language 287
Multiplexing and Demultiplexing 289
Layers Forever? 290

Copyrighted material

18 PYTHON'S DICTIONARY IMPLEMENTATION: BEING ALL THINGS TO ALL PEOPLE 293
by Andrew Kuchling

Inside the Dictionary 295
Special Accommodations 296
Collisions 298
Resizing 299
Iterations and Dynamic Changes 300
Conclusion 301
Acknowledgments 301

19 MULTIDIMENSIONAL ITERATORS IN NUMPY 303
by Travis E. Oliphant

Key Challenges in N-Dimensional Array Operations 304
Memory Models for an N-Dimensional Array 305
NumPy Iterator Origins 307
Iterator Design 307
Iterator Interface 313
Iterator Use 314
Conclusion 318

20 A HIGHLY RELIABLE ENTERPRISE SYSTEM FOR NASA'S MARS ROVER MISSION 319
by Ronald Mak

The Mission and the Collaborative Information Portal 320
Mission Needs 321
System Architecture 322
Case Study: The Streamer Service 325
Reliability 328
Robustness 336
Conclusion 338

21 ERP5: DESIGNING FOR MAXIMUM ADAPTABILITY 339
by Rogerio Atem de Carvalho and Rafael Monnerat

General Goals of ERP 340
ERP5 340
The Underlying Zope Platform 342
ERP5 Project Concepts 346
Coding the ERP5 Project 347
Conclusion 351

Copyrighted material

22 A SPOONFUL OF SEWAGE 353
by Bryan Cantrill

23 DISTRIBUTED PROGRAMMING WITH MAPREDUCE 371
by Jeffrey Dean and Sanjay Ghemawat

A Motivating Example 371
The MapReduce Programming Model 374
Other MapReduce Examples 375
A Distributed MapReduce Implementation 377
Extensions to the Model 380
Conclusion 381
Further Reading 381
Acknowledgments 382
Appendix: Word Count Solution 382

24 BEAUTIFUL CONCURRENCY 385
by Simon Peyton Jones

A Simple Example: Bank Accounts 386
Software Transactional Memory 388
The Santa Claus Problem 396
Reflections on Haskell 404
Conclusion 404
Acknowledgments 406

25 SYNTACTIC ABSTRACTION: THE SYNTAX-CASE EXPANDER 407
by R. Kent Dybvig

Brief Introduction to syntax-case 411
Expansion Algorithm 413
Example 425
Conclusion 428

26 LABOR-SAVING ARCHITECTURE: AN OBJECT-ORIENTED FRAMEWORK FOR NETWORKED SOFTWARE 429
by William R. Otte and Douglas C. Schmidt

Sample Application: Logging Service 431
Object-Oriented Design of the Logging Server Framework 433
Implementing Sequential Logging Servers 439
Implementing Concurrent Logging Servers 444
Conclusion 450

27 INTEGRATING BUSINESS PARTNERS THE RESTFUL WAY 451
by Andrew Patzer

Project Background 452
Exposing Services to External Clients 452
Routing the Service Using the Factory Pattern 456
Exchanging Data Using E-Business Protocols 457
Conclusion 462

Copyrighted material

28 BEAUTIFUL DEBUGGING 463
by Andreas Zeller

Debugging a Debugger 464
A Systematic Process 466
A Search Problem 467
Finding the Failure Cause Automatically 468
Delta Debugging 470
Minimizing Input 472
Hunting the Defect 473
A Prototype Problem 475
Conclusion 476
Acknowledgments 476
Further Reading 476

29 TREATING CODE AS AN ESSAY 477
by Yukihiro Matsumoto

30 WHEN A BUTTON IS ALL THAT CONNECTS YOU TO THE WORLD 483
by Arun Mehta

Basic Design Model 484
Input Interface 487
Efficiency of the User Interface 500
Download 500
Future Directions 500

31 EMACSPEAK: THE COMPLETE AUDIO DESKTOP 503
by T. V. Raman

Producing Spoken Output 504
Speech-Enabling Emacs 505
Painless Access to Online Information 516
Summary 522
Acknowledgments 525

32 CODE IN MOTION 527
by Laura Wingerd and Christopher Seiwald

On Being "Bookish" 528
Alike Looking Alike 529
The Perils of Indentation 530
Navigating Code 531
The Tools We Use 532
DiffMerge's Checkered Past 534
Conclusion 536
Acknowledgments 536
Further Reading 536

Copyrighted material

33 WRITING PROGRAMS FOR "THE BOOK" 539
by Brian Hayes

The Nonroyal Road 540
Warning to Parenthophobes 540
Three in a Row 541
The Slippery Slope 544
The Triangle Inequality 545
Meandering On 547
"Duh!"—I Mean "Aha!" 548
Conclusion 550
Further Reading 550

AFTERWORD 553
by Andy Oram

CONTRIBUTORS 555

INDEX 565

Copyrighted material

Foreword

Greg Wilson

I GOT MY FIRST JOB AS A PROGRAMMER IN THE SUMMER OF 1982. Two weeks after I started, one of the system administrators loaned me Kernighan and Plauger's *The Elements of Programming Style* (McGraw-Hill) and Wirth's *Algorithms + Data Structures = Programs* (Prentice Hall). They were a revelation—for the first time, I saw that programs could be more than just instructions for computers. They could be as elegant as well-made kitchen cabinets, as graceful as a suspension bridge, or as eloquent as one of George Orwell's essays.

Time and again since that summer, I have heard people bemoan the fact that our profession doesn't teach students to see this. Architects are taught to look at buildings, and composers study one another's scores, but programmers—they look at each other's work only when there's a bug to fix; even then, they try to look at as little as possible. We tell students to use sensible variable names, introduce them to some basic design patterns, and then wonder why so much of what they write is so ugly.

This book is our attempt to fix this. In May 2006, I asked some well-known (and not so well-known) software designers to dissect and discuss the most beautiful piece of code they knew. As this book shows, they have found beauty in many different places. For

Copyrighted material

some, it lives in the small details of elegantly crafted software. Others find beauty in the big picture—in how a program's structure allows it to evolve gracefully over time, or in the techniques used to build it.

Wherever they find it, I am grateful to our contributors for taking time to give us a tour. I hope that you enjoy reading this book as much as Andy and I have enjoyed editing it, and that it inspires you to create something beautiful, too.

Copyrighted material

Preface

BEAUTIFUL CODE WAS CONCEIVED BY GREG WILSON IN 2006 as a way to elicit insights from leading software developers and computer scientists. Together, he and his co-editor, Andy Oram, approached experts with diverse backgrounds from all over the world. They received a flood of responses, partly because royalties from the book are being donated to Amnesty International. The results of the project appear in this volume.

As wide-ranging as this book is, it represents just a small fraction of what is happening in this most exciting of fields. Thousand of other projects, equally interesting and educational, are being moved forward every day by other programmers whom we did not contact. Furthermore, many excellent practitioners who were asked for chapters do not appear in this book because they were too busy at the time, preferred not to contribute to Amnesty International, or had conflicting obligations. To benefit from the insights of all these people, we hope to do further books along similar lines in the future.

How This Book Is Organized

Chapter 1, *A Regular Expression Matcher*, by Brian Kernighan, shows how deep insight into a language and a problem can lead to a concise and elegant solution.

Copyrighted material

Chapter 2, *Subversion's Delta Editor: Interface As Ontology*, by Karl Fogel, starts with a well-chosen abstraction and demonstrates its unifying effects on the system's further development.

Chapter 3, *The Most Beautiful Code I Never Wrote*, by Jon Bentley, suggests how to measure a procedure without actually executing it.

Chapter 4, *Finding Things*, by Tim Bray, draws together many strands in Computer Science in an exploration of a problem that is fundamental to many computing tasks.

Chapter 5, *Correct, Beautiful, Fast (in That Order): Lessons from Designing XML Verifiers*, by Elliotte Rusty Harold, reconciles the often conflicting goals of thoroughness and good performance.

Chapter 6, *Framework for Integrated Test: Beauty Through Fragility*, by Michael Feathers, presents an example that breaks the rules and achieves its own elegant solution.

Chapter 7, *Beautiful Tests*, by Alberto Savoia, shows how a broad, creative approach to testing can not only eliminate bugs but turn you into a better programmer.

Chapter 8, *On-the-Fly Code Generation for Image Processing*, by Charles Petzold, drops down a level to improve performance while maintaining portability.

Chapter 9, *Top Down Operator Precedence*, by Douglas Crockford, revives an almost forgotten parsing technique and shows its new relevance to the popular JavaScript language.

Chapter 10, *The Quest for an Accelerated Population Count*, by Henry S. Warren, Jr., reveals the impact that some clever algorithms can have on even a seemingly simple problem.

Chapter 11, *Secure Communication: The Technology Of Freedom*, by Ashish Gulhati, discusses the directed evolution of a secure messaging application that was designed to make sophisticated but often confusing cryptographic technology intuitively accessible to users.

Chapter 12, *Growing Beautiful Code in BioPerl*, by Lincoln Stein, shows how the combination of a flexible language and a custom-designed module can make it easy for people with modest programming skills to create powerful visualizations for their data.

Chapter 13, *The Design of the Gene Sorter*, by Jim Kent, combines simple building blocks to produce a robust and valuable tool for gene researchers.

Chapter 14, *How Elegant Code Evolves with Hardware: The Case of Gaussian Elimination*, by Jack Dongarra and Piotr Luszczek, surveys the history of LINPACK and related major software packages to show how assumptions must constantly be re-evaluated in the face of new computing architectures.

Chapter 15, *The Long-Term Benefits of Beautiful Design*, by Adam Kolawa, explains how attention to good design principles many decades ago helped CERN's widely used mathematical library (the predecessor of LINPACK) stand the test of time.

Copyrighted material

Chapter 16, *The Linux Kernel Driver Model: The Benefits of Working Together*, by Greg Kroah-Hartman, explains how many efforts by different collaborators to solve different problems led to the successful evolution of a complex, multithreaded system.

Chapter 17, *Another Level of Indirection*, by Diomidis Spinellis, shows how the flexibility and maintainability of the FreeBSD kernel is promoted by abstracting operations done in common by many drivers and filesystem modules.

Chapter 18, *Python's Dictionary Implementation: Being All Things to All People*, by Andrew Kuchling, explains how a careful design combined with accommodations for a few special cases allows a language feature to support many different uses.

Chapter 19, *Multidimensional Iterators in NumPy*, by Travis E. Oliphant, takes you through the design steps that succeed in hiding complexity under a simple interface.

Chapter 20, *A Highly Reliable Enterprise System for NASA's Mars Rover Mission*, by Ronald Mak, uses industry standards, best practices, and Java technologies to meet the requirements of a NASA expedition where reliability cannot be in doubt.

Chapter 21, *ERP5: Designing for Maximum Adaptability*, by Rogerio Atem de Carvalho and Rafael Monnerat, shows how a powerful ERP system can be developed with free software tools and a flexible architecture.

Chapter 22, *A Spoonful of Sewage*, by Bryan Cantrill, lets the reader accompany the author through a hair-raising bug scare and a clever solution that violated expectations.

Chapter 23, *Distributed Programming with MapReduce*, by Jeff Dean and Sanjay Ghemawat, describes a system that provides an easy-to-use programming abstraction for large-scale distributed data processing at Google that automatically handles many difficult aspects of distributed computation, including automatic parallelization, load balancing, and failure handling.

Chapter 24, *Beautiful Concurrency*, by Simon Peyton Jones, removes much of the difficulty of parallel programs through Software Transactional Memory, demonstrated here using Haskell.

Chapter 25, *Syntactic Abstraction: The syntax-case Expander*, by R. Kent Dybvig, shows how macros—a key feature of many languages and systems—can be protected in Scheme from producing erroneous output.

Chapter 26, *Labor-Saving Architecture: An Object-Oriented Framework for Networked Software*, by William R. Otte and Douglas C. Schmidt, applies a range of standard object-oriented design techniques, such as patterns and frameworks, to distributed logging to keep the system flexible and modular.

Chapter 27, *Integrating Business Partners the RESTful Way*, by Andrew Patzer, demonstrates a designer's respect for his programmers by matching the design of a B2B web service to its requirements.

Copyrighted material

Chapter 28, *Beautiful Debugging*, by Andreas Zeller, shows how a disciplined approach to validating code can reduce the time it takes to track down errors.

Chapter 29, *Treating Code As an Essay*, by Yukihiro Matsumoto, lays out some challenging principles that drove his design of the Ruby programming language, and that, by extension, will help produce better software in general.

Chapter 30, *When a Button Is All That Connects You to the World*, by Arun Mehta, takes you on a tour through the astounding interface design choices involved in a text-editing system that allows people with severe motor disabilities, like Professor Stephen Hawking, to communicate via a computer.

Chapter 31, *Emacspeak: The Complete Audio Desktop*, by T. V. Raman, shows how Lisp's advice facility can be used with Emacs to address a general need—generating rich spoken output—that cuts across all aspects of the Emacs environment, without modifying the underlying source code of a large software system.

Chapter 32, *Code in Motion*, by Laura Wingerd and Christopher Seiwald, lists some simple rules that have unexpectedly strong impacts on programming accuracy.

Chapter 33, *Writing Programs for "The Book"*, by Brian Hayes, explores the frustrations of solving a seemingly simple problem in computational geometry, and its surprising resolution.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, mathematical variables, URLs, file and directory names, and commands.

`Constant width`
: Indicates elements of program code, the contents of files, and text output displayed on a computer console.

`Constant width bold`
: Shows commands or other text typed literally by the user.

`Constant width italic`
: Shows text replaced with user-supplied values.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books *does* require permission.

Copyrighted material

Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation *does* require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Beautiful Code*, edited by Andy Oram and Greg Wilson. Copyright 2007 O'Reilly Media, Inc., 978-0-596-51004-6."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://www.oreilly.com/catalog/9780596510046

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our web site at:

http://www.oreilly.com

Safari® Enabled

When you see a Safari® Enabled icon on the cover of your favorite technology book, that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top tech books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at *http://safari.oreilly.com*.

Copyrighted material

Copyrighted material

CHAPTER ONE

A Regular Expression Matcher

Brian Kernighan

REGULAR EXPRESSIONS ARE NOTATIONS FOR DESCRIBING PATTERNS OF TEXT and, in effect, make up a special-purpose language for pattern matching. Although there are myriad variants, all share the idea that most characters in a pattern match literal occurrences of themselves, but some *metacharacters* have special meaning, such as * to indicate some kind of repetition or `[...]` to mean any one character from the set within the brackets.

In practice, most searches in programs such as text editors are for literal words, so the regular expressions are often literal strings like `print`, which will match `printf` or `sprint` or `printer paper` anywhere. In so-called *wildcards* used to specify filenames in Unix and Windows, a * matches any number of characters, so the pattern `*.c` matches all filenames that end in `.c`. There are many, many variants of regular expressions, even in contexts where one would expect them to be the same. Jeffrey Friedl's *Mastering Regular Expressions* (O'Reilly) is an exhaustive study of the topic.

Stephen Kleene invented regular expressions in the mid-1950s as a notation for finite automata; in fact, they are equivalent to finite automata in what they represent. They first appeared in a program setting in Ken Thompson's version of the QED text editor in the mid-1960s. In 1967, Thompson applied for a patent on a mechanism for rapid text matching based on regular expressions. The patent was granted in 1971, one of the very first software patents [U.S. Patent 3,568,156, Text Matching Algorithm, March 2, 1971].

Copyrighted material

Regular expressions moved from QED to the Unix editor *ed*, and then to the quintessential Unix tool *grep*, which Thompson created by performing radical surgery on *ed*. These widely used programs helped regular expressions become familiar throughout the early Unix community.

Thompson's original matcher was very fast because it combined two independent ideas. One was to generate machine instructions on the fly during matching so that it ran at machine speed rather than by interpretation. The other was to carry forward all possible matches at each stage, so it did not have to backtrack to look for alternative potential matches. In later text editors that Thompson wrote, such as *ed*, the matching code used a simpler algorithm that backtracked when necessary. In theory, this is slower, but the patterns found in practice rarely involved backtracking, so the *ed* and *grep* algorithm and code were good enough for most purposes.

Subsequent regular expression matchers like *egrep* and *fgrep* added richer classes of regular expressions, and focused on fast execution no matter what the pattern. Ever-fancier regular expressions became popular and were included not only in C-based libraries, but also as part of the syntax of scripting languages such as Awk and Perl.

The Practice of Programming

In 1998, Rob Pike and I were writing *The Practice of Programming* (Addison-Wesley). The last chapter of the book, "Notation," collected a number of examples where good notation led to better programs and better programming. This included the use of simple data specifications (`printf`, for instance), and the generation of code from tables.

Because of our Unix backgrounds and nearly 30 years of experience with tools based on regular expression notation, we naturally wanted to include a discussion of regular expressions, and it seemed mandatory to include an implementation as well. Given our emphasis on tools, it also seemed best to focus on the class of regular expressions found in *grep*—rather than, say, those from shell wildcards—since we could also then talk about the design of *grep* itself.

The problem was that any existing regular expression package was far too big. The local *grep* was over 500 lines long (about 10 book pages) and encrusted with barnacles. Open source regular expression packages tended to be huge—roughly the size of the entire book—because they were engineered for generality, flexibility, and speed; none were remotely suitable for pedagogy.

I suggested to Rob that we find the smallest regular expression package that would illustrate the basic ideas while still recognizing a useful and nontrivial class of patterns. Ideally, the code would fit on a single page.

Rob disappeared into his office. As I remember it now, he emerged in no more than an hour or two with the 30 lines of C code that subsequently appeared in Chapter 9 of *The Practice of Programming*. That code implements a regular expression matcher that handles the following constructs.

Copyrighted material

Character	Meaning
c	Matches any literal character *c*.
. (period)	Matches any single character.
^	Matches the beginning of the input string.
$	Matches the end of the input string.
*	Matches zero or more occurrences of the previous character.

This is quite a useful class; in my own experience of using regular expressions on a day-to-day basis, it easily accounts for 95 percent of all instances. In many situations, solving the right problem is a big step toward creating a beautiful program. Rob deserves great credit for choosing a very small yet important, well-defined, and extensible set of features from among a wide set of options.

Rob's implementation itself is a superb example of beautiful code: compact, elegant, efficient, and useful. It's one of the best examples of recursion that I have ever seen, and it shows the power of C pointers. Although at the time we were most interested in conveying the important role of good notation in making a program easier to use (and perhaps easier to write as well), the regular expression code has also been an excellent way to illustrate algorithms, data structures, testing, performance enhancement, and other important topics.

Implementation

In *The Practice of Programming*, the regular expression matcher is part of a standalone program that mimics *grep*, but the regular expression code is completely separable from its surroundings. The main program is not interesting here; like many Unix tools, it reads either its standard input or a sequence of files, and prints those lines that contain a match of the regular expression.

This is the matching code:

```
/* match: search for regexp anywhere in text */
int match(char *regexp, char *text)
{
    if (regexp[0] == '^')
        return matchhere(regexp+1, text);
    do {    /* must look even if string is empty */
        if (matchhere(regexp, text))
            return 1;
    } while (*text++ != '\0');
    return 0;
}

/* matchhere: search for regexp at beginning of text */
int matchhere(char *regexp, char *text)
{
    if (regexp[0] == '\0')
        return 1;
    if (regexp[1] == '*')
        return matchstar(regexp[0], regexp+2, text);
```


Copyrighted material

```
    if (regexp[0] == '$' && regexp[1] == '\0')
        return *text == '\0';
    if (*text!='\0' && (regexp[0]=='.' || regexp[0]==*text))
        return matchhere(regexp+1, text+1);
    return 0;
}

/* matchstar: search for c*regexp at beginning of text */
int matchstar(int c, char *regexp, char *text)
{
    do {    /* a * matches zero or more instances */
        if (matchhere(regexp, text))
            return 1;
    } while (*text != '\0' && (*text++ == c || c == '.'));
    return 0;
}
```

Discussion

The function `match(regexp, text)` tests whether there is an occurrence of the regular expression anywhere within the text; it returns 1 if a match is found and 0 if not. If there is more than one match, it finds the leftmost and shortest.

The basic operation of `match` is straightforward. If the first character of the regular expression is ^ (an anchored match), any possible match must occur at the beginning of the string. That is, if the regular expression is `^xyz`, it matches `xyz` only if `xyz` occurs at the beginning of the text, not somewhere in the middle. This is tested by matching the rest of the regular expression against the text starting at the beginning and nowhere else. Otherwise, the regular expression might match anywhere within the string. This is tested by matching the pattern against each character position of the text in turn. If there are multiple matches, only the first (leftmost) one will be identified. That is, if the regular expression is `xyz`, it will match the first occurrence of `xyz` regardless of where it occurs.

Notice that advancing over the input string is done with a `do-while` loop, a comparatively unusual construct in C programs. The occurrence of a `do-while` instead of a `while` should always raise a question: why isn't the loop termination condition being tested at the beginning of the loop, before it's too late, rather than at the end after something has been done? But the test is correct here: since the * operator permits zero-length matches, we first have to check whether a null match is possible.

The bulk of the work is done in the function `matchhere(regexp, text)`, which tests whether the regular expression matches the text that begins right here. The function `matchhere` operates by attempting to match the first character of the regular expression with the first character of the text. If the match fails, there can be no match at this text position and `matchhere` returns 0. If the match succeeds, however, it's possible to advance to the next character of the regular expression and the next character of the text. This is done by calling `matchhere` recursively.

Copyrighted material

The situation is a bit more complicated because of some special cases, and of course the need to stop the recursion. The easiest case is that if the regular expression is at its end (`regexp[0] == '\0'`), all previous tests have succeeded, and thus the regular expression matches the text.

If the regular expression is a character followed by a *, `matchstar` is called to see whether the closure matches. The function `matchstar(c, regexp, text)` tries to match repetitions of the text character `c`, beginning with zero repetitions and counting up, until it either finds a match of the rest of the text, or it fails and thus concludes that there is no match. This algorithm identifies a "shortest match," which is fine for simple pattern matching as in *grep*, where all that matters is finding a match as quickly as possible. A "longest match" is more intuitive and almost certain to be better for a text editor where the matched text will be replaced. Most modern regular expression libraries provide both alternatives, and *The Practice of Programming* presents a simple variant of `matchstar` for this case, shown below.

If the regular expression consists of a `$` at the end of the expression, the text matches only if it too is at its end:

```
if (regexp[0] == '$' && regexp[1] == '\0')
    return *text == '\0';
```

Otherwise, if we are not at the end of the text string (that is, `*text!='\0'`), and if the first character of the text string matches the first character of the regular expression, so far so good; we go on to test whether the next character of the regular expression matches the next character of the text by making a recursive call to `matchhere`. This recursive call is the heart of the algorithm and the reason why the code is so compact and clean.

If all of these attempts to match fail, there can be no match at this point between the regular expression and the text, so `matchhere` returns 0.

This code uses C pointers intensively. At each stage of the recursion, if something matches, the recursive call that follows uses pointer arithmetic (e.g., `regexp+1` and `text+1`) so that the subsequent function is called with the next character of the regular expression and of the text. The depth of recursion is no more than the length of the pattern, which in normal use is quite short, so there is no danger of running out of space.

Alternatives

This is a very elegant and well-written piece of code, but it's not perfect. What might we do differently? I might rearrange `matchhere` to deal with `$` before `*`. Although it makes no difference here, it feels a bit more natural, and a good rule is to do easy cases before difficult ones.

In general, however, the order of tests is critical. For instance, in this test from `matchstar`:

```
} while (*text != '\0' && (*text++ == c || c == '.'));
```

we must advance over one more character of the text string no matter what, so the increment in `text++` must always be performed.

Copyrighted material

This code is careful about termination conditions. Generally, the success of a match is determined by whether the regular expression runs out at the same time as the text does. If they do run out together, that indicates a match; if one runs out before the other, there is no match. This is perhaps most obvious in a line like:

```
if (regexp[0] == '$' && regexp[1] == '\0')
    return *text == '\0';
```

but subtle termination conditions show up in other cases as well.

The version of `matchstar` that implements leftmost longest matching begins by identifying a maximal sequence of occurrences of the input character `c`. Then it uses `matchhere` to try to extend the match to the rest of the pattern and the rest of the text. Each failure reduces the number of `c`s by one and tries again, including the case of zero occurrences:

```
/* matchstar: leftmost longest search for c*regexp */
int matchstar(int c, char *regexp, char *text)
{
    char *t;

    for (t = text; *t != '\0' && (*t == c || c == '.'); t++)
        ;
    do { /* * matches zero or more */
        if (matchhere(regexp, t))
            return 1;
    } while (t-- > text);
    return 0;
}
```

Consider the regular expression `(.*)`, which matches arbitrary text within parentheses. Given the target text:

```
for (t = text; *t != '\0' && (*t == c || c == '.'); t++)
```

a longest match from the beginning will identify the entire parenthesized expression, while a shortest match will stop at the first right parenthesis. (Of course a longest match beginning from the second left parenthesis will extend to the end of the text.)

Building on It

The purpose of *The Practice of Programming* was to teach good programming. At the time the book was written, Rob and I were still at Bell Labs, so we did not have firsthand experience of how the book would be best used in a classroom. It has been gratifying to discover that some of the material does work well in classes. I have used this code since 2000 as a vehicle for teaching important points about programming.

First, it shows how recursion is useful and leads to clean code in a novel setting; it's not yet another version of Quicksort (or factorial!), nor is it some kind of tree walk.

It's also a good example for performance experiments. Its performance is not very different from the system versions of *grep*, which shows that the recursive technique is not too costly and that it's not worth trying to tune the code.

Copyrighted material

On the other hand, it is also a fine illustration of the importance of a good algorithm. If a pattern includes several `.*` sequences, the straightforward implementation requires a lot of backtracking, and, in some cases, will run very slowly indeed.

The standard Unix *grep* has the same backtracking properties. For example, the command:

```
grep 'a.*a.*a.*a.a'
```

takes about 20 seconds to process a 4 MB text file on a typical machine.

An implementation based on converting a nondeterministic finite automaton to a deterministic automaton, as in *egrep*, will have much better performance on hard cases; it can process the same pattern and the same input in less than one-tenth of a second, and running time in general is independent of the pattern.

Extensions to the regular expression class can form the basis of a variety of assignments. For example:

1. Add other metacharacters, such as `+` for one or more occurrences of the previous character, or `?` for zero or one matches. Add some way to quote metacharacters, such as `\$` to stand for a literal occurrence of `$`.
2. Separate regular expression processing into a *compilation* phase and an *execution* phase. Compilation converts the regular expression into an internal form that makes the matching code simpler or allows the subsequent matching to run faster. This separation is not necessary for the simple class of regular expressions in the original design, but it makes sense in *grep*-like applications where the class is richer and the same regular expression is used for a large number of input lines.
3. Add character classes such as `[abc]` and `[0-9]`, which in conventional *grep* notation match `a` or `b` or `c` and a digit, respectively. This can be done in several ways, the most natural of which seems to be replacing the `char*` variables of the original code with a structure:

```
typedef struct RE {
        int     type;   /* CHAR, STAR, etc. */
        int     ch;     /* the character itself */
        char    *ccl;   /* for [...] instead */
        int     nccl;   /* true if class is negated [^...] */
} RE;
```

   and modifying the basic code to handle an array of these instead of an array of characters. It's not strictly necessary to separate compilation from execution for this situation, but it turns out to be a lot easier. Students who follow the advice to precompile into such a structure invariably do better than those who try to interpret some complicated pattern data structure on the fly.

   Writing clear and unambiguous specifications for character classes is tough, and implementing them perfectly is worse, requiring a lot of tedious and uninstructive coding. I have simplified this assignment over time, and today most often ask for Perl-like shorthands such as `\d` for digit and `\D` for nondigit instead of the original bracketed ranges.

Copyrighted material

4. Use an opaque type to hide the RE structure and all the implementation details. This is a good way to show object-oriented programming in C, which doesn't support much beyond this. In effect, this creates a regular expression class that uses function names like `RE_new()` and `RE_match()` for the methods instead of the syntactic sugar of an object-oriented language.
5. Modify the class of regular expressions to be like the wildcards in various shells: matches are implicitly anchored at both ends, * matches any number of characters, and ? matches any single character. One can modify the algorithm or map the input into the existing algorithm.
6. Convert the code to Java. The original code uses C pointers very well, and it's good practice to figure out the alternatives in a different language. Java versions use either `String.charAt` (indexing instead of pointers) or `String.substring` (closer to the pointer version). Neither seems as clear as the C code, and neither is as compact. Although performance isn't really part of this exercise, it is interesting to see that the Java implementation runs roughly six or seven times slower than the C versions.
7. Write a wrapper class that converts from this class's regular expressions to Java's Pattern and Matcher classes, which separate the compilation and matching in a quite different way. This is a good example of the Adapter or Facade pattern, which puts a different face on an existing class or set of functions.

I've also used this code extensively to explore testing techniques. Regular expressions are rich enough that testing is far from trivial, but small enough that one can quickly write down a substantial collection of tests to be performed mechanically. For extensions like those just listed, I ask students to write a large number of tests in a compact language (yet another example of "notation") and use those tests on their own code; naturally, I use their tests on other students' code as well.

Conclusion

I was amazed by how compact and elegant this code was when Rob Pike first wrote it—it was much smaller and more powerful than I had thought possible. In hindsight, one can see a number of reasons why the code is so small.

First, the features are well chosen to be the most useful and to give the most insight into implementation, without any frills. For example, the implementation of the anchored patterns ^ and $ requires only three or four lines, but it shows how to deal with special cases cleanly before handling the general cases uniformly. The closure operation * must be present because it is a fundamental notion in regular expressions and provides the only way to handle patterns of unspecified lengths. But it would add no insight to also provide + and ?, so those are left as exercises.

Copyrighted material

Second, recursion is a win. This fundamental programming technique almost always leads to smaller, cleaner, and more elegant code than the equivalent written with explicit loops, and that is the case here. The idea of peeling off one matching character from the front of the regular expression and from the text, then recursing for the rest, echoes the recursive structure of the traditional factorial or string length examples, but in a much more interesting and useful setting.

Third, this code really uses the underlying language to good effect. Pointers can be misused, of course, but here they are used to create compact expressions that naturally express the extracting of individual characters and advancing to the next character. Array indexing or substrings can achieve the same effect, but in this code, pointers do a better job, especially when coupled with C idioms for autoincrement and implicit conversion of truth values.

I don't know of another piece of code that does so much in so few lines while providing such a rich source of insight and further ideas.

Copyrighted material

Copyrighted material

CHAPTER TWO

Subversion's Delta Editor: Interface As Ontology

Karl Fogel

EXAMPLES OF BEAUTIFUL CODE TEND TO BE LOCAL SOLUTIONS to well-bounded, easily comprehensible problems, such as Duff's Device (*http://en.wikipedia.org/wiki/Duff's_device*) or *rsync*'s rolling checksum algorithm (*http://en.wikipedia.org/wiki/Rsync#Algorithm*). This is not because small, simple solutions are the only beautiful kind, but because appreciating complex code requires more context than can be given on the back of a napkin.

Here, with the luxury of several pages to work in, I'd like to talk about a larger sort of beauty—not necessarily the kind that would strike a passing reader immediately, but the kind that programmers who work with the code on a regular basis would come to appreciate as they accumulate experience with the problem domain. My example is not an algorithm, but an interface: the programming interface used by the open source version control system Subversion (*http://subversion.tigris.org*) to express the difference between two directory trees, which is also the interface used to transform one tree into the other. In Subversion, its formal name is the C type `svn_delta_editor_t`, but it is known colloquially as the *delta editor*.

Copyrighted material

Subversion's delta editor demonstrates the properties that programmers look for in good design. It breaks down the problem along boundaries so natural that anyone designing a new feature for Subversion can easily tell when to call each function, and for what purpose. It presents the programmer with uncontrived opportunities to maximize efficiency (such as by eliminating unnecessary data transfers over the network) and allows for easy integration of auxiliary tasks (such as progress reporting). Perhaps most important, the design has proved very resilient during enhancements and updates.

And as if to confirm suspicions about the origins of good design, the delta editor was created by a single person over the course of a few hours (although that person was very familiar with the problem and the code base).

To understand what makes the delta editor beautiful, we must start by examining the problem it solves.

Version Control and Tree Transformation

Very early in the Subversion project, the team realized we had a general task that would be performed over and over: that of minimally expressing the difference between two similar (usually related) directory trees. As a version control system, one of Subversion's goals is to track revisions to directory structures as well as individual file contents. In fact, Subversion's server-side repository is fundamentally designed around directory versioning. A repository is simply a series of snapshots of a directory tree as that tree transforms over time. For each changeset committed to the repository, a new tree is created, differing from the preceding tree exactly where the changes are located and nowhere else. The unchanged portions of the new tree share storage with the preceding tree, and so on back into time. Each successive version of the tree is labeled with a monotonically increasing integer; this unique identifier is called a *revision number*.

Think of the repository as an array of revision numbers, stretching off into infinity. By convention, revision 0 is always an empty directory. In Figure 2-1, revision 1 has a tree hanging off it (typically the initial import of content into the repository), and no other revisions have been committed yet. The boxes represent nodes in this virtual filesystem: each node is either a directory (labeled DIR in the upper-right corner) or a file (labeled FILE).

What happens when we modify *tuna*? First, we make a new file node, containing the latest text. The new node is not connected to anything yet. As Figure 2-2 shows, it's just hanging out there in space, with no name.

Next, we create a new revision of its parent directory. As Figure 2-3 shows, the subgraph is still not connected to the revision array.

Copyrighted material

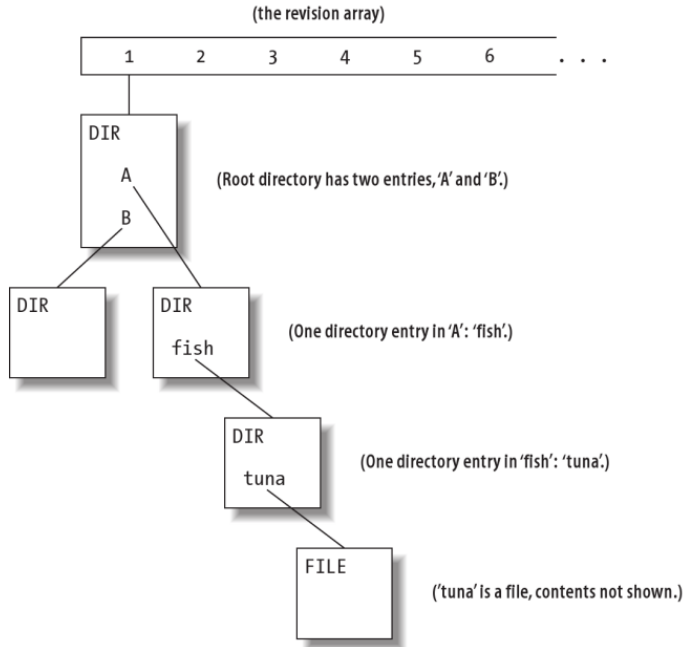


FIGURE 2-1. Conceptual view of revision numbers

FIGURE 2-2. New node when just created

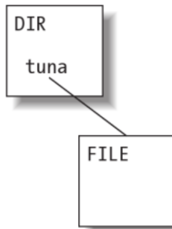


FIGURE 2-3. Creation of new parent directory

We continue up the line, creating a new revision of the next parent directory (Figure 2-4).

At the top, we create a new revision of the root directory, as shown in Figure 2-5. This new directory needs an entry to point to the "new" directory A, but since directory B hasn't changed at all, the new root directory also has an entry still pointing to the *old* directory B's node.

Copyrighted material

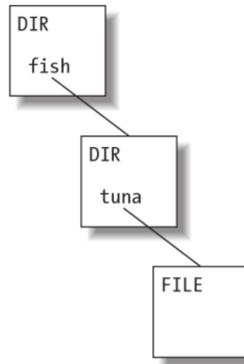


FIGURE 2-4. Continuing to move up, creating parent directories

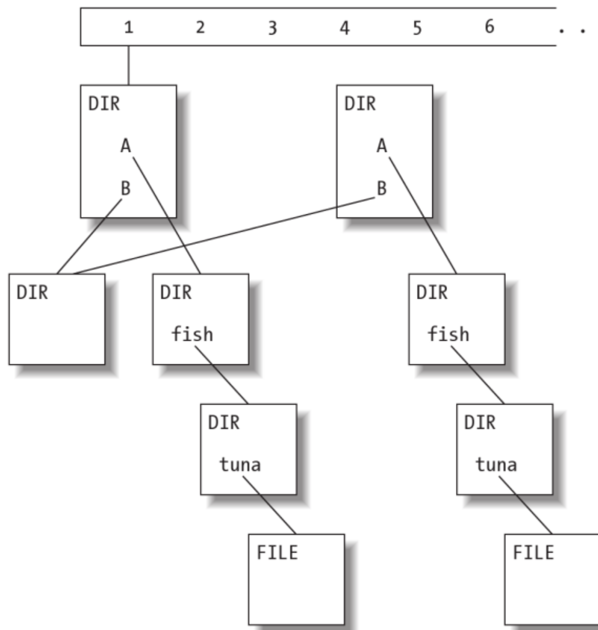


FIGURE 2-5. Complete new directory tree

Now that all the new nodes are written, we finish the "bubble up" process by linking the new tree to the next available revision in the history array, thus making it visible to repository users (Figure 2-6). In this case, the new tree becomes revision 2.

Thus each revision in the repository points to the root node of a unique tree, and the difference between that tree and the preceding one is the change that was committed in the new revision. To trace the changes, a program walks down both trees simultaneously, noting where entries point to different places. (For brevity, I've left out some details, such as saving storage space by compressing older nodes as differences against their newer versions.)

Copyrighted material

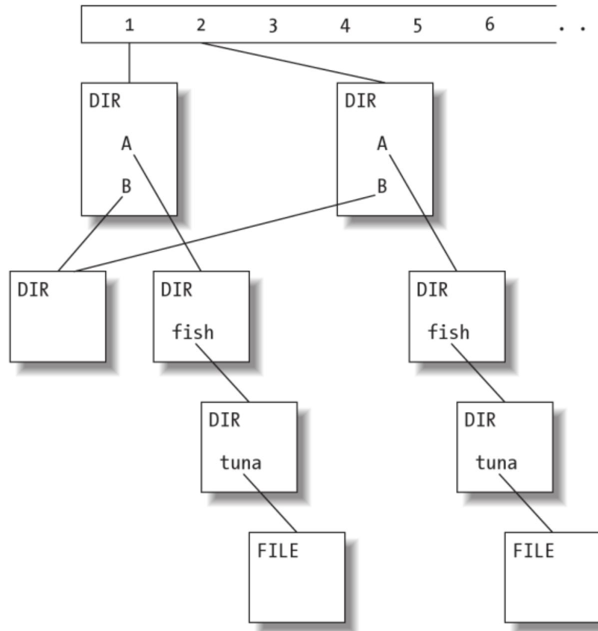


FIGURE 2-6. Finished revision: link to new tree

Although this tree-versioning model is all background to the main point of this chapter (the delta editor, which we'll come to soon), it has such nice properties that I considered making it the subject of its own chapter, as an example of beautiful code. Some of its attractive features are:

Easy reads
: To locate revision *n* of file */path/to/foo.txt*, one jumps to revision *n*, then walks down the tree to */path/to/foo.txt*.

Writers don't interfere with readers
: As writers create new nodes, bubbling their way up to the top, concurrent readers cannot see the work in progress. A new tree becomes visible to readers only after the writer makes its final "link" to a revision number in the repository.

Tree structure is versioned
: The very structure of each tree is being saved from revision to revision. File and directory renames, additions, and deletions become an intrinsic part of the repository's history.

If Subversion were only a repository, this would be the end of the story. However, there's a client side, too: the *working copy*, which is a user's checked-out copy of some revision tree plus whatever local edits the user has made but not yet committed. (Actually, working copies do not always reflect a single revision tree; they often contain mixtures of nodes from different revisions. This turns out not to make much of a difference as far as tree transformation is concerned. So, for the purposes of this chapter, just assume that a working copy represents some revision tree, though not necessarily that of the latest revision.)

Copyrighted material

Expressing Tree Differences

The most common action in Subversion is to transmit changes between the two sides: from the repository to the working copy when doing an update to receive others' changes, and from the working copy to the repository when committing one's own changes. Expressing the difference between two trees is also key to many other common operations—e.g., diffing, switching to a branch, merging changes from one branch to another, and so on.

Clearly it would be silly to have two different interfaces, one for server → client and another for client → server. The underlying task is the same in both cases. A tree difference is a tree difference, regardless of which direction it's traveling over the network or what its consumer intends to do with it. But finding a natural way to express tree differences proved surprisingly challenging. Complicating matters further, Subversion supports multiple network protocols and multiple backend storage mechanisms; we needed an interface that would look the same across all of those.

Our initial attempts to come up with an interface ranged from unsatisfying to downright awkward. I won't describe them all here, but what they had in common was that they tended to leave open questions for which there were no persuasive answers.

For example, many of the solutions involved transmitting the changed paths as strings, either as full paths or path components. Well, what order should the paths be transmitted in? Depth first? Breadth first? Random order? Alphabetical? Should there be different commands for directories than for files? Most importantly, how would each individual command expressing a difference know that it was part of a larger operation grouping all the changes into a unified set? In Subversion, the concept of the overall tree operation is quite user-visible, and if the programmatic interface didn't intrinsically match that concept, we'd surely need to write lots of brittle glue code to compensate.

In frustration, I drove with another developer, Ben Collins-Sussman, from Chicago down to Bloomington, Indiana, to seek the advice of Jim Blandy, who had invented Subversion's repository model in the first place, and who has, shall we say, strong opinions about design. Jim listened quietly as we described the various avenues we'd explored for transmitting tree differences, his expression growing grimmer and grimmer as we talked. When we reached the end of the list, he sat for a moment and then politely asked us to scram for a while so he could think. I put on my jogging shoes and went running; Ben stayed behind and read a book in another room or something. So much for collaborative development.

After I returned from my run and showered, Ben and I went back into Jim's den, and he showed us what he'd come up with. It is essentially what's in Subversion today; there have been various changes over the years, but none to its fundamental structure.

Copyrighted material

The Delta Editor Interface

Following is a mildly abridged version of the delta editor interface. I've left out the parts that deal with copying and renaming, the parts related to Subversion properties (properties are versioned metadata, and are not important here), and parts that handle some other Subversion-specific bookkeeping. However, you can always see the latest version of the delta editor by visiting *http://svn.collab.net/repos/svn/trunk/subversion/include/svn_delta.h*. This chapter is based on r21731 (that is, revision 21731) at *http://svn.collab.net/viewvc/svn/trunk/subversion/include/svn_delta.h?revision=21731*.

To understand the interface, even in its abridged form, you'll need to know some Subversion jargon:

pools
: The `pool` arguments are memory pools—allocation buffers that allow a large number of objects to be freed simultaneously.

`svn_error_t`
: The return type `svn_error_t` simply means that the function returns a pointer to a Subversion error object; a successful call returns a null pointer.

text delta
: A text delta is the difference between two different versions of a file; you can apply a text delta as a patch to one version of the file to produce the other version. In Subversion, the "text" of a file is considered binary data—it doesn't matter whether the file is plain text, audio data, an image, or something else. Text deltas are expressed as streams of fixed-sized windows, each window containing a chunk of binary diff data. This way, peak memory usage is proportional to the size of a single window, rather than to the total size of the patch (which might be quite large in the case of, say, an image file).

window handler
: This is the function prototype for applying one window of text-delta data to a target file.

baton
: This is a `void *` data structure that provides context to a callback function. In other APIs, these are sometimes called `void *ctx`, `void *userdata`, or `void *closure`. Subversion calls them "batons" because they're passed around a lot, like batons in a relay race.

The interface starts with an introduction, to put a reader of the code in the right frame of mind. This text is almost unchanged since Jim Blandy wrote it in August of 2000, so the general concept has clearly weathered well:

```
/** Traversing tree deltas.
 *
 * In Subversion, we've got various producers and consumers of tree
 * deltas.
 *
```


Copyrighted material

```
 * In processing a `commit' command:
 * - The client examines its working copy data, and produces a tree
 *   delta describing the changes to be committed.
 * - The client networking library consumes that delta, and sends them
 *   across the wire as an equivalent series of network requests.
 * - The server receives those requests and produces a tree delta ---
 *   hopefully equivalent to the one the client produced above.
 * - The Subversion server module consumes that delta and commits an
 *   appropriate transaction to the filesystem.
 *
 * In processing an `update' command, the process is reversed:
 * - The Subversion server module talks to the filesystem and produces
 *   a tree delta describing the changes necessary to bring the
 *   client's working copy up to date.
 * - The server consumes this delta, and assembles a reply
 *   representing the appropriate changes.
 * - The client networking library receives that reply, and produces a
 *   tree delta --- hopefully equivalent to the one the Subversion
 *   server produced above.
 * - The working copy library consumes that delta, and makes the
 *   appropriate changes to the working copy.
 *
 * The simplest approach would be to represent tree deltas using the
 * obvious data structure.  To do an update, the server would
 * construct a delta structure, and the working copy library would
 * apply that structure to the working copy; the network layer's job
 * would simply be to get the structure across the net intact.
 *
 * However, we expect that these deltas will occasionally be too large
 * to fit in a typical workstation's swap area.  For example, in
 * checking out a 200Mb source tree, the entire source tree is
 * represented by a single tree delta.  So it's important to handle
 * deltas that are too large to fit in swap all at once.
 *
 * So instead of representing the tree delta explicitly, we define a
 * standard way for a consumer to process each piece of a tree delta
 * as soon as the producer creates it.  The svn_delta_editor_t
 * structure is a set of callback functions to be defined by a delta
 * consumer, and invoked by a delta producer.  Each invocation of a
 * callback function describes a piece of the delta --- a file's
 * contents changing, something being renamed, etc.
 */
```

Then comes a long, formal documentation comment, followed by the interface itself, which is a callback table whose invocation order is partially constrained:

```
/** A structure full of callback functions the delta source will invoke
 * as it produces the delta.
 *
 * Function Usage
 * ==============
 *
 * Here's how to use these functions to express a tree delta.
 *
```


Copyrighted material

```
* The delta consumer implements the callback functions described in
* this structure, and the delta producer invokes them.  So the
* caller (producer) is pushing tree delta data at the callee
* (consumer).
*
* At the start of traversal, the consumer provides edit_baton, a
* baton global to the entire delta edit.
*
* Next, if there are any tree deltas to express, the producer should
* pass the edit_baton to the open_root function, to get a baton
* representing root of the tree being edited.
*
* Most of the callbacks work in the obvious way:
*
*    delete_entry
*    add_file
*    add_directory
*    open_file
*    open_directory
*
* Each of these takes a directory baton, indicating the directory
* in which the change takes place, and a path argument, giving the
* path (relative to the root of the edit) of the file,
* subdirectory, or directory entry to change. Editors will usually
* want to join this relative path with some base stored in the edit
* baton (e.g. a URL, a location in the OS filesystem).
*
* Since every call requires a parent directory baton, including
* add_directory and open_directory, where do we ever get our
* initial directory baton, to get things started?  The open_root
* function returns a baton for the top directory of the change.  In
* general, the producer needs to invoke the editor's open_root
* function before it can get anything of interest done.
*
* While open_root provides a directory baton for the root of
* the tree being changed, the add_directory and open_directory
* callbacks provide batons for other directories.  Like the
* callbacks above, they take a parent_baton and a relative path
* path, and then return a new baton for the subdirectory being
* created / modified --- child_baton.  The producer can then use
* child_baton to make further changes in that subdirectory.
*
* So, if we already have subdirectories named `foo' and `foo/bar',
* then the producer can create a new file named `foo/bar/baz.c' by
* calling:
*
*    - open_root () --- yielding a baton root for the top directory
*
*    - open_directory (root, "foo") --- yielding a baton f for `foo'
*
*    - open_directory (f, "foo/bar") --- yielding a baton b for `foo/bar'
*
*    - add_file (b, "foo/bar/baz.c")
*
```


Copyrighted material

```
 * When the producer is finished making changes to a directory, it
 * should call close_directory.  This lets the consumer do any
 * necessary cleanup, and free the baton's storage.
 *
 * The add_file and open_file callbacks each return a baton
 * for the file being created or changed.  This baton can then be
 * passed to apply_textdelta to change the file's contents.
 * When the producer is finished making changes to a file, it should
 * call close_file, to let the consumer clean up and free the baton.
 *
 * Function Call Ordering
 * ======================
 *
 * There are five restrictions on the order in which the producer
 * may use the batons:
 *
 * 1. The producer may call open_directory, add_directory,
 *    open_file, add_file at most once on any given directory
 *    entry. delete_entry may be called at most once on any given
 *    directory entry and may later be followed by add_directory or
 *   add_file on the same directory entry. delete_entry may
 *    not be called on any directory entry after open_directory,
 *   add_directory,open_file or add_file has been called on
 *    that directory entry.
 *
 * 2. The producer may not close a directory baton until it has
 *    closed all batons for its subdirectories.
 *
 * 3. When a producer calls open_directory or add_directory,
 *    it must specify the most recently opened of the currently open
 *    directory batons.  Put another way, the producer cannot have
 *    two sibling directory batons open at the same time.
 *
 * 4. When the producer calls open_file or add_file, it must
 *    follow with any changes to the file (using apply_textdelta),
 *    followed by a close_file call, before issuing any other
 *    file or directory calls.
 *
 * 5. When the producer calls apply_textdelta, it must make all of
 *    the window handler calls (including the NULL window at the
 *    end) before issuing any other svn_delta_editor_t calls.
 *
 * So, the producer needs to use directory and file batons as if it
 * is doing a single depth-first traversal of the tree.
 *
 * Pool Usage
 * ==========
 *
 * Many editor functions are invoked multiple times, in a sequence
 * determined by the editor "driver". The driver is responsible for
 * creating a pool for use on each iteration of the editor function,
 * and clearing that pool between each iteration. The driver passes
 * the appropriate pool on each function invocation.
 *
```


Copyrighted material

```
 * Based on the requirement of calling the editor functions in a
 * depth-first style, it is usually customary for the driver to similarly
 * nest the pools. However, this is only a safety feature to ensure
 * that pools associated with deeper items are always cleared when the
 * top-level items are also cleared. The interface does not assume, nor
 * require, any particular organization of the pools passed to these
 * functions.
 */
typedef struct svn_delta_editor_t
{
  /** Set *root_baton to a baton for the top directory of the change.
   * (This is the top of the subtree being changed, not necessarily
   * the root of the filesystem.)  Like any other directory baton, the
   * producer should call close_directory on root_baton when they're
   * done.
   */
  svn_error_t *(*open_root)(void *edit_baton,
                            apr_pool_t *dir_pool,
                            void **root_baton);


  /** Remove the directory entry named path, a child of the directory
   * represented by parent_baton.
   */
  svn_error_t *(*delete_entry)(const char *path,
                               void *parent_baton,
                               apr_pool_t *pool);


  /** We are going to add a new subdirectory named path.  We will use
   * the value this callback stores in *child_baton as the
   * parent_baton for further changes in the new subdirectory.
   */
  svn_error_t *(*add_directory)(const char *path,
                                void *parent_baton,
                                apr_pool_t *dir_pool,
                                void **child_baton);

  /** We are going to make changes in a subdirectory (of the directory
   * identified by parent_baton). The subdirectory is specified by
   * path. The callback must store a value in *child_baton that
   * should be used as the parent_baton for subsequent changes in this
   * subdirectory.
   */
  svn_error_t *(*open_directory)(const char *path,
                                 void *parent_baton,
                                 apr_pool_t *dir_pool,
                                 void **child_baton);

  /** We are done processing a subdirectory, whose baton is dir_baton
   * (set by add_directory or open_directory).  We won't be using
   * the baton any more, so whatever resources it refers to may now be
   * freed.
   */
  svn_error_t *(*close_directory)(void *dir_baton,
                                  apr_pool_t *pool);
```


Copyrighted material

```
    /** We are going to add a new file named path.  The callback can
     * store a baton for this new file in **file_baton; whatever value
     * it stores there should be passed through to apply_textdelta.
     */
    svn_error_t *(*add_file)(const char *path,
                             void *parent_baton,
                             apr_pool_t *file_pool,
                             void **file_baton);

    /** We are going to make change to a file named path, which resides
     * in the directory identified by parent_baton.
     *
     * The callback can store a baton for this new file in **file_baton;
     * whatever value it stores there should be passed through to
     * apply_textdelta.
     */
    svn_error_t *(*open_file)(const char *path,
                              void *parent_baton,
                              apr_pool_t *file_pool,
                              void **file_baton);

    /** Apply a text delta, yielding the new revision of a file.
     *
     * file_baton indicates the file we're creating or updating, and the
     * ancestor file on which it is based; it is the baton set by some
     * prior add_file or open_file callback.
     *
     * The callback should set *handle to a text delta window
     * handler; we will then call *handle on successive text
     * delta windows as we receive them.  The callback should set
     * *handler_baton to the value we should pass as the baton
     * argument to *handler.
     */
    svn_error_t *(*apply_textdelta)(void *file_baton,
                                    apr_pool_t *pool,
                                    svn_txdelta_window_handler_t *handler,
                                    void **handler_baton);

    /** We are done processing a file, whose baton is file_baton (set by
     * add_file or open_file).  We won't be using the baton any
     * more, so whatever resources it refers to may now be freed.
     */
    svn_error_t *(*close_file)(void *file_baton,
                               apr_pool_t *pool);

    /** All delta processing is done.  Call this, with the edit_baton for
     * the entire edit.
     */
    svn_error_t *(*close_edit)(void *edit_baton,
                               apr_pool_t *pool);

    /** The editor-driver has decided to bail out.  Allow the editor to
     * gracefully clean up things if it needs to.
     */
    svn_error_t *(*abort_edit)(void *edit_baton,
                               apr_pool_t *pool);

  } svn_delta_editor_t;
```


Copyrighted material

But Is It Art?

I cannot claim that the beauty of this interface was immediately obvious to me. I'm not sure it was obvious to Jim either; he was probably just trying to get Ben and me out of his house. But he'd been pondering the problem for a long time, too, and he followed his instincts about how tree structures behave.

The first thing that strikes one about the delta editor is that it *chooses* constraint: even though there is no philosophical requirement that tree edits be done in depth-first order (or indeed in any order at all), the interface enforces depth-firstness anyway, by means of the baton relationships. This makes the interface's usage and behavior more predictable.

The second thing is that an entire edit operation unobtrusively carries its context with it, again by means of the batons. A file baton can contain a pointer to its parent directory baton, a directory baton can contain a pointer to *its* parent directory baton (with a null parent for the root of the edit), and everyone can contain a pointer to the global edit baton. Although an individual baton may be a disposable object—for example, when a file is closed, its baton is destroyed—any given baton allows access to the global edit context, which may contain, for example, the revision number the client side is being updated to. Thus, batons are overloaded: they provide scope (i.e., lifetime, because a baton only lasts as long as the pool in which it is allocated) to portions of the edit, but they also carry global context.

The third important feature is that the interface provides clear boundaries between the various suboperations involved in expressing a tree change. For example, opening a file merely indicates that something changed in that file between the two trees, but doesn't give details; calling `apply_textdelta` gives the details, but you don't have to call `apply_textdelta` if you don't want to. Similarly, opening a directory indicates that something changed in or under that directory, but if you don't need to say any more than that, you can just close the directory and move on. These boundaries are a consequence of the interface's dedication to *streaminess*, as expressed in its introductory comment: *"…instead of representing the tree delta explicitly, we define a standard way for a consumer to process each piece of a tree delta as soon as the producer creates it."* It would have been tempting to stream only the largest data chunks (that is, the file diffs), but the delta editor interface goes the whole way and streams the entire tree delta, thus giving both producer and consumer fine-grained control over memory usage, progress reporting, and interruptibility.

It was only after we began throwing the new delta editor at various problems that these features began to show their value. For example, one thing we wanted to implement was change summarization: a way to show an overview of the difference between two trees without giving the details. This is useful when someone wants to know which files in her working copy have changed in the repository since she checked them out, but doesn't need to know exactly what the changes were.

Copyrighted material

Here's a slightly simplified version of how it works: the client tells the server what revision tree the working copy is based on, and then the server tells the client the difference between that revision tree and the latest one, using the delta editor. The server is the producer, the client is the consumer.

Using the repository from earlier in the chapter, in which we built up a change to */A/fish/tuna* to create revision 2, let's see how this would look as a series of editor calls, sent by the server to a client whose tree is still at revision 1. The `if` block about two-thirds of the way through is where we decide whether this is a summarization edit or a "give me everything" edit:

```
svn_delta_editor_t *editor
void *edit_baton;

/* In real life, this would be a passed-in parameter, of course. */
int summarize_only = TRUE;

/* In real life, these variables would be declared in subroutines,
   so that their lifetimes would be bound to the stack frame just
   as the objects they point to are bound by the tree edit frame. */
void *root_baton;
void *dir_baton;
void *subdir_baton;
void *file_baton;

/* Similarly, there would be subpools, not just one top-level pool. */
apr_pool_t *pool = svn_pool_create();

/* Each use of the delta editor interface starts by requesting the
   particular editor that implements whatever you need, e.g.,
   streaming the edit across the network, applying it to a working
   copy, etc. */
Get_Update_Editor(&editor, &eb,
                  some_repository,
                  1, /* source revision number */
                  2, /* target revision number */
                  pool);

/* Now we drive the editor. In real life, this sequence of calls
   would be dynamically generated, by code that walks the two
   repository trees and invokes editor->foo() as appropriate. */

editor->open_root(edit_baton, pool, &root_baton);
editor->open_directory("A", root_baton, pool, &dir_baton);
editor->open_directory("A/fish", dir_baton, pool, &subdir_baton);
editor->open_file("A/fish/tuna", subdir_baton, pool, &file_baton);

if (! summarize_only)
  {
    svn_txdelta_window_handler_t window_handler;
    void *window_handler_baton;
    svn_txdelta_window_t *window;

    editor->apply_textdelta(file_baton, pool
                            apr_pool_t *pool,
                            &window_handler,
                            &window_handler_baton);
```


Copyrighted material

```
        do {
          window = Get_Next_TextDelta_Window(...);
          window_handler(window, window_handler_baton);
        } while (window);
      }

    editor->close_file(file_baton, pool);
    editor->close_directory(subdir_baton, pool);
    editor->close_directory(dir_baton, pool);
    editor->close_directory(root_baton, pool);
    editor->close_edit(edit_baton, pool);
```

As this example shows, the distinction between a summary of a change and the full version of the change falls naturally along the boundaries of the delta editor interface, allowing us to use the same code path for both purposes. While it happens that the two revision trees in this example were adjacent (revision 1 and revision 2), they didn't have to be. The same method would work for any two trees, even with many revisions between them, as is the case when a working copy hasn't been updated for a long time. And it would work when the two trees are in reverse order—that is, with the newer revision first. This is useful for reverting a change.

Abstraction As a Spectator Sport

Our next indication of the delta editor's flexibility came when we needed to do two or more distinct things in the same tree edit. One of the earliest such situations was the need to handle cancellations. When the user interrupted an update, a signal handler trapped the request and set a flag; then at various points during the operation, we checked the flag and exited cleanly if it was set. It turned out that in most cases, the safest place to exit the operation was simply the next entry or exit boundary of an editor function call. This was trivially true for operations that performed no I/O on the client side (such as change summarizations and diffs), but it was also true of many operations that did touch files. After all, most of the work in an update is simply writing out the data, and even if the user interrupts the overall update, it usually still makes sense to either finish writing or cleanly cancel whatever file was in progress when the interrupt was detected.

But where to implement the flag checks? We could hardcode them into the delta editor, the one returned (by reference) from `Get_Update_Editor()`. But that's obviously a poor choice: the delta editor is a library function that might be called from code that wants a totally different style of cancellation checking, or none at all.

A slightly better solution would be to pass a cancellation-checking callback function and associated baton to `Get_Update_Editor()`. The returned editor would periodically invoke the callback on the baton and, depending on the return value, either continue as normal or return early (if the callback is null, it is never invoked). But that arrangement isn't ideal either. Checking cancellation is really a parasitic goal: you might want to do it when updating, or you might not, but in any case it has no effect on the way the update process itself works. Ideally, the two shouldn't be tangled together in the code, especially as we had concluded that, for the most part, operations didn't need fine-grained control over cancellation checking, anyway—the editor call boundaries would do just fine.

Copyrighted material

Cancellation is just one example of an auxiliary task associated with tree delta edits. We faced, or thought we faced, similar problems in keeping track of committed targets while transmitting changes from the client to the server, in reporting update or commit progress to the user, and in various other situations. Naturally, we looked for a way to abstract out these adjunct behaviors, so the core code wouldn't be cluttered with them. In fact, we looked so hard that we initially *over*-abstracted:

```
/** Compose editor_1 and its baton with editor_2 and its baton.
 *
 * Return a new editor in new_editor (allocated in pool), in which
 * each function fun calls editor_1->fun and then editor_2->fun,
 * with the corresponding batons.
 *
 * If editor_1->fun returns error, that error is returned from
 * new_editor->fun and editor_2->fun is never called; otherwise
 * new_editor->fun's return value is the same as editor_2->fun's.
 *
 * If an editor function is null, it is simply never called, and this
 * is not an error.
 */
void
svn_delta_compose_editors(const svn_delta_editor_t **new_editor,
                          void **new_edit_baton,
                          const svn_delta_editor_t *editor_1,
                          void *edit_baton_1,
                          const svn_delta_editor_t *editor_2,
                          void *edit_baton_2,
                          apr_pool_t *pool);
```

Although this turned out to go a bit too far—we'll look at why in a moment—I still find it a testament to the beauty of the editor interface. The composed editors behaved predictably, they kept the code clean (because no individual editor function had to worry about the details of some parallel editor invoked before or after it), and they passed the associativity test: you could take an editor that was itself the result of a composition and compose it with other editors, and everything would *just work*. It worked because the editors all agreed on the basic shape of the operation they were performing, even though they might do totally different things with the data.

As you can tell, I still miss editor composition for its sheer elegance. But in the end it was more abstraction than we needed. Much of the functionality that we initially implemented using composed editors, we later rewrote to use custom callbacks passed to the editor-creation routines. Although the adjunct behaviors did usually line up with editor call boundaries, they often weren't appropriate at *all* call boundaries, or even at most of them. The result was an overly high infrastructure-to-work ratio: by setting up an entire parallel editor, we were misleadingly implying to readers of the code that the adjunct behaviors would be invoked more often than they actually were.

Copyrighted material

Having gone as far as we could with editor composition and then retreated, we were still free to implement it by hand when we really wanted it, however. Today in Subversion, cancellation is done with manual composition. The cancellation-checking editor's constructor takes another editor—the core operation editor—as a parameter:

```
/** Set *editor and *edit_baton to a cancellation editor that
 * wraps wrapped_editor and wrapped_baton.
 *
 * The editor will call cancel_func with cancel_baton when each of
 * its functions is called, continuing on to call the corresponding wrapped
 * function if cancel_func returns SVN_NO_ERROR.
 *
 * If cancel_func is NULL, set *editor to wrapped_editor and
 * *edit_baton to wrapped_baton.
 */
svn_error_t *
svn_delta_get_cancellation_editor(svn_cancel_func_t cancel_func,
                                  void *cancel_baton,
                                  const svn_delta_editor_t *wrapped_editor,
                                  void *wrapped_baton,
                                  const svn_delta_editor_t **editor,
                                  void **edit_baton,
                                  apr_pool_t *pool);
```

We also implement some conditional debugging traces using a similar process of manual composition. The other adjunct behaviors—primarily progress reporting, event notification, and target counting—are implemented via callbacks that are passed to the editor constructors and (if nonnull) invoked by the editor at the few places where they are needed.

The editor interface continues to provide a strong unifying force across Subversion's code. It may seem strange to praise an API that first tempted its users into over-abstraction, but that temptation was mainly a side effect of suiting the problem of streamy tree delta transmission exceptionally well—it made the problem look so tractable that we wanted other problems to become that problem! When they didn't fit, we backed off, but the editor constructors still provided a canonical place to inject callbacks, and the editor's internal operation boundaries helped guide our thinking about when to invoke those callbacks.

Conclusions

The real strength of this API, and, I suspect, of any good API, is that it guides one's thinking. All operations involving tree modification in Subversion are now shaped roughly the same way. This not only reduces the amount of time newcomers must spend learning existing code, it gives new code a clear model to follow, and developers have taken the hint. For example, the *svnsync* feature, which mirrors one repository's activity directly into another repository—and was added to Subversion in 2006, six years after the advent of the delta editor—uses the delta editor interface to transmit the activity. The developer of

Copyrighted material

that feature was not only spared the need to design a change-transmission mechanism, he was spared the need to even *consider* whether he needed to design a change-transmission mechanism. And others who now hack on the new code find it feels mostly familiar the first time they see it.

These are significant benefits. Not only does having the right API reduce learning time, it also relieves the development community of the need to have certain debates: design discussions that would have spawned long and contentious mailing list threads simply do not come up. That may not be quite the same thing as pure technical or aesthetic beauty, but in a project with many participants and a constant turnover rate, it's a beauty you can use.

Copyrighted material

CHAPTER THREE

The Most Beautiful Code I Never Wrote

Jon Bentley

I ONCE HEARD A MASTER PROGRAMMER PRAISED WITH THE PHRASE, "He adds function by deleting code." Antoine de Saint-Exupéry, the French writer and aviator, expressed this sentiment more generally when he said, "A designer knows he has achieved perfection not when there is nothing left to add, but when there is nothing left to take away." In software, the most beautiful code, the most beautiful functions, and the most beautiful programs are sometimes not there at all.

It is, of course, difficult to talk about things that aren't there. This chapter attempts this daunting task by presenting a novel analysis of the runtime of the classic Quicksort program. The first section sets the stage by reviewing Quicksort from a personal perspective. The subsequent section is the meat of this chapter. We'll start by adding one counter to the program, then manipulate the code to make it smaller and smaller and yet more and more powerful until just a few lines of code completely capture its average runtime. The third section summarizes the techniques, and presents a particularly succinct analysis of the cost of binary search trees. The final two sections draw insights from the chapter to help you write more elegant programs.

Copyrighted material

The Most Beautiful Code I Ever Wrote

When Greg Wilson first described the idea of this book, I asked myself what was the most beautiful code I had ever written. After this delicious question rolled around my brain for the better part of a day, I realized that the answer was easy: Quicksort. Unfortunately, the one question has three different answers, depending on precisely how it is phrased.

I wrote my thesis on divide-and-conquer algorithms, and found that C.A.R. Hoare's Quicksort ("Quicksort," *Computer Journal* 5) is undeniably the granddaddy of them all. It is a beautiful algorithm for a fundamental problem that can be implemented in elegant code. I loved the algorithm, but I always tiptoed around its innermost loop. I once spent two days debugging a complex program that was based on that loop, and for years I carefully copied that code whenever I needed to perform a similar task. It solved my problems, but I didn't *really* understand it.

I eventually learned an elegant partitioning scheme from Nico Lomuto, and was finally able to write a Quicksort that I could understand and even prove correct. William Strunk Jr.'s observation that "vigorous writing is concise" applies to code as well as to English, so I followed his admonition to "omit needless words" (*The Elements of Style*). I finally reduced approximately 40 lines of code to an even dozen. So if the question is, "What is the most beautiful small piece of code that you've ever written?" my answer is the Quicksort from my book *Programming Pearls*, Second Edition (Addison-Wesley). This Quicksort function, implemented in C, is shown in Example 3-1. We'll further study and refine this example in the next section.

EXAMPLE 3-1. Quicksort function

```
void quicksort(int l, int u)
{   int i, m;
    if (l >= u) return;
    swap(l, randint(l, u));
    m = l;
    for (i = l+1; i <= u; i++)
        if (x[i] < x[l])
            swap(++m, i);
    swap(l, m);
    quicksort(l, m-1);
    quicksort(m+1, u);
}
```

This code sorts a global array `x[n]` when called with the arguments `quicksort(0, n-1)`. The two parameters of the function are the indexes of the subarray to be sorted: `l` for lower and `u` for upper. The call `swap(i,j)` exchanges the contents of `x[i]` and `x[j]`. The first swap randomly chooses a partitioning element uniformly selected between `l` and `u`.

Programming Pearls contains a detailed derivation and proof of correctness for the `quicksort` function. Throughout the rest of this chapter, I'll assume that the reader is familiar with Quicksort to the level presented in that description and in most elementary algorithms textbooks.

Copyrighted material

If you change the question to, "What is the most beautiful piece of code that you've written that was widely used?" my answer is again a Quicksort. An article I wrote with M. D. McIlroy ("Engineering a sort function," *Software–Practice and Experience*, Vol. 23, No. 11) describes a serious performance bug in the venerable Unix `qsort` function. We set out to build a new C library `sort` function, and considered many different algorithms for the task, including Merge Sort and Heap Sort. After comparing several possible implementations, we settled on a version of the Quicksort algorithm. That paper describes how we engineered a new function that was clearer, faster, and more robust than its competitors—partly because it was smaller. Gordon Bell's sage advice proved true: "The cheapest, fastest, and most reliable components of a computer system are those that aren't there." That function has now been widely used for over a decade with no reports of failure.

Considering the gains that could be achieved by reducing code size, I finally asked myself a third variant of the question that began this chapter. "What is the most beautiful code that you *never* wrote?" How was I able to accomplish a great deal with very little? The answer was once again related to Quicksort, specifically, the analysis of its performance. The next section tells that tale.

More and More with Less and Less

Quicksort is an elegant algorithm that lends itself to subtle analysis. Around 1980, I had a wonderful discussion with Tony Hoare about the history of his algorithm. He told me that when he first developed Quicksort, he thought it was too simple to publish, and only wrote his classic "Quicksort" paper after he was able to analyze its expected runtime.

It is easy to see that in the worst case, Quicksort might take about n^2 time to sort an array of n elements. In the best case, it chooses the median value as a partitioning element, and therefore sorts an array in about $n \lg n$ comparisons. So, how many comparisons does it use on the average for a random array of n distinct values?

Hoare's analysis of this question is beautiful, but unfortunately over the mathematical heads of many programmers. When I taught Quicksort to undergraduates, I was frustrated that many just didn't "get" the proof, even after sincere effort. We'll now attack that problem experimentally. We'll start with Hoare's program, and eventually end up with an analysis close to his.

Our task is to modify Example 3-1 of the randomizing Quicksort code to analyze the average number of comparisons used to sort an array of distinct inputs. We will also attempt to gain maximum insight with minimal code, runtime, and space.

To determine the average number of comparisons, we first augment the program to count them. To do this, we increment the variable `comps` before the comparison in the inner loop (Example 3-2).

Copyrighted material

EXAMPLE 3-2. Quicksort inner loop instrumented to count comparisons

```
for (i = l+1; i <= u; i++) {
    comps++;
    if (x[i] < x[l])
        swap(++m, i);
}
```

If we run the program for one value of n, we'll see how many comparisons that particular run takes. If we repeat that for many runs over many values of n, and analyze the results statistically, we'll observe that, on average, Quicksort takes about $1.4\, n \lg n$ comparisons to sort n elements.

That isn't a bad way to gain insight into the behavior of a program. Thirteen lines of code and a few experiments can reveal a lot. A famous quote attributed to writers such as Blaise Pascal and T. S. Eliot states that, "If I had more time, I would have written you a shorter letter." We have the time, so let's experiment with the code to attempt to create a shorter (and better) program.

We'll play the game of speeding up that experiment, trying to increase statistical accuracy and programming insight. Because the inner loop always makes precisely u-l comparisons, we can make the program a tiny bit faster by counting those comparisons in a single operation outside the loop. This change yields the Quicksort shown in Example 3-3.

EXAMPLE 3-3. Quicksort inner loop with increment moved out of loop

```
comps += u-l;
for (i = l+1; i <= u; i++)
    if (x[i] < x[l])
        swap(++m, i);
```

This program sorts an array and counts the number of comparisons used while doing so. However, if our goal is only to count the comparisons, we don't really need to sort the array. Example 3-4 removes the "real work" of sorting the elements, and keeps only the "skeleton" of the various calls made by the program.

EXAMPLE 3-4. Quicksort skeleton reduced to counting

```
void quickcount(int l, int u)
{   int m;
    if (l >= u) return;
    m = randint(l, u);
    comps += u-l;
    quickcount(l, m-1);
    quickcount(m+1, u);
}
```

This program works because of the "randomizing" way in which Quicksort chooses its partitioning element, and because all of the elements are assumed to be distinct. This new program now runs in time proportional to n, and while Example 3-3 required space proportional to n, the space is now reduced to the recursion stack, which on average is proportional to $\lg n$.

Copyrighted material

While the indexes (l and *u*) of the array are critical in an actual program, they don't matter in this skeleton version. We can replace these two indexes with a single integer (n) that specifies the size of the subarray to be sorted (see Example 3-5).

EXAMPLE 3-5. Quicksort skeleton with single size argument

```
void qc(int n)
{   int m;
    if (n <= 1) return;
    m = randint(1, n);
    comps += n-1;
    qc(m-1);
    qc(n-m);
}
```

It is now more natural to rephrase this procedure as a *comparison count* function that returns the number of comparisons used by one random Quicksort run. This function is shown in Example 3-6.

EXAMPLE 3-6. Quicksort skeleton implemented as a function

```
int cc(int n)
{   int m;
    if (n <= 1) return 0;
    m = randint(1, n);
    return n-1 + cc(m-1) + cc(n-m);
}
```

Examples 3-4, 3-5, and 3-6 all solve the same basic problem, and they do so with the same runtime and memory usage. Each successor improves the form of the function and is thereby clearer and a bit more succinct than its predecessor.

In defining the *inventor's paradox* (*How To Solve It*, Princeton University Press), George Pólya says that "the more ambitious plan may have more chances of success." We will now try to exploit that paradox in the analysis of Quicksort. So far we have asked, "How many comparisons does Quicksort make on one run of size *n*?" We will now ask the more ambitious question, "How many comparisons does Quicksort make, on average, for a random array of size *n*?" We can extend Example 3-6 to yield the pseudocode in Example 3-7.

EXAMPLE 3-7. Quicksort average comparisons as pseudocode

```
float c(int n)
    if (n <= 1) return 0
    sum = 0
    for (m = 1; m <= n; m++)
        sum += n-1 + c(m-1) + c(n-m)
    return sum/n
```

If the input has a maximum of one element, Quicksort uses no comparisons, just as in Example 3-6. For larger *n*, this code considers each partition value *m* (from the first element to the last, each equally likely) and determines the cost of partitioning there. The code then calculates the sum of these values (thereby solving recursively one problem of size m-1 and one problem of size *n-m*), and then divides that sum by *n* to return the average.

Copyrighted material

If we could compute this number it would make our experiments much more powerful. Rather than having to run many experiments at a single value of n to estimate the mean, a single experiment would give us the true mean. Unfortunately, that power comes at a price: the program runs in time proportional to 3^n (it is an interesting, if self-referential, exercise to analyze that runtime using the techniques described in this chapter).

Example 3-7 takes the time that it does because it computes subanswers over and over again. When a program does that, we can often use *dynamic programming* to store the subanswers to avoid recomputing them. In this case, we'll introduce the table `t[N+1]`, in which `t[n]` stores `c(n)`, and compute its values in increasing order. We will let `N` denote the maximum size of `n`, which is the size of the array to be sorted. The result is shown in Example 3-8.

EXAMPLE 3-8. Quicksort calculation with dynamic programming

```
t[0] = 0
for (n = 1; n <= N; n++)
    sum = 0
    for (i = 1; i <= n; i++)
        sum += n-1 + t[i-1] + t[n-i]
    t[n] = sum/n
```

This program is a rough transcription of Example 3-7 and replaces `c(n)` with `t[n]`. Its runtime is proportional to N^2 and its space is proportional to `N`. One of its benefits is that at the end of execution, the array `t` contains the true average values (not just the estimate of sample means) for array elements `0` through `N`. Those values can be analyzed to yield insight about the functional form of the expected number of comparisons used by Quicksort.

We will now simplify that program further. The first step is to move the term `n-1` out of the loop, as shown in Example 3-9.

EXAMPLE 3-9. Quicksort calculation with code moved out of the loop

```
t[0] = 0
for (n = 1; n <= N; n++)
    sum = 0
    for (i = 1; i <= n; i++)
        sum += t[i-1] + t[n-i]
    t[n] = n-1 +  sum/n
```

We will now further tune the loop by exploiting symmetry. When `n` is 4, for instance, the inner loop computes the sum:

```
t[0]+t[3] + t[1]+t[2] + t[2]+t[1] + t[3]+t[0]
```

In the sequence of pairs, the first elements increase while the second elements decrease. We can therefore rewrite the sum as:

```
2 * (t[0] + t[1] + t[2] + t[3])
```

We can use that symmetry to yield the Quicksort shown in Example 3-10.

Copyrighted material

EXAMPLE 3-10. Quicksort calculation with symmetry

```
t[0] = 0
for (n = 1; n <= N; n++)
    sum = 0
    for (i = 0; i < n; i++)
        sum += 2 * t[i]
    t[n] = n-1 + sum/n
```

However, this code is once again wasteful because it recomputes the same sum over and over again. Rather than adding all the previous terms, we can initialize `sum` outside the loop and add the next term to yield Example 3-11.

EXAMPLE 3-11. Quicksort calculation with the inner loop removed

```
sum = 0; t[0] = 0
for (n = 1; n <= N; n++)
    sum += 2*t[n-1]
    t[n] = n-1 + sum/n
```

This little program is indeed useful. In time proportional to *N*, it produces a table of the true expected runtimes of Quicksort for every integer from 1 to *N*.

Example 3-11 is straightforward to implement in a spreadsheet, where the values are immediately made available for further analysis. Table 3-1 shows the first rows.

TABLE 3-1. Output of spreadsheet implementation of Example 3-11

N	Sum	t[n]
0	0	0
1	0	0
2	0	1
3	2	2.667
4	7.333	4.833
5	17	7.4
6	31.8	10.3
7	52.4	13.486
8	79.371	16.921

The first row of numbers in this table is initialized with the three constants from the code. In spreadsheet notation, the next row of numbers (the third row of the spreadsheet) is calculated using the following relations:

```
A3 = A2+1      B3 = B2 + 2*C2      C3 = A3-1 + B3/A3
```

Dragging those (relative) relations down completes the spreadsheet. That spreadsheet is a real contender for "the most beautiful code I ever wrote," using the criterion of accomplishing a great deal with just a few lines of code.

But what if we don't need all the values? What if we would prefer to analyze just a few of the values along the way (for example, all the powers of 2 from 2^0 to 2^{32})? Although Example 3-11 builds the complete table `t`, it uses only the most recent value of that table.

Copyrighted material

We can therefore replace the linear space of the table t[] with the constant space of the variable t, as shown in Example 3-12.

EXAMPLE 3-12. Quicksort calculation—final version

```
sum = 0; t = 0
for (n = 1; n <= N; n++)
    sum += 2*t
    t = n-1 + sum/n
```

We could then insert an extra line of code to test for appropriateness of n, and print those results as needed.

This tiny program is the final step in our long path. Alan Perlis' observation is apt in consideration of the path this chapter has taken: "Simplicity does not precede complexity, but follows it" ("Epigrams on Programming," *Sigplan Notices*, Vol. 17, Issue 9).

Perspective

Table 3-2 summarizes the programs used to analyze Quicksort throughout this chapter.

TABLE 3-2. Evolution of Quicksort comparison counting

Example number	Lines of code	Type of answer	Number of answers	Runtime	Space
2	13	Sample	1	$n \lg n$	N
3	13	"	"	"	"
4	8	"	"	n	$\lg n$
5	8	"	"	"	"
6	6	"	"	"	"
7	6	Exact	"	3^N	N
8	6	"	N	N^2	N
9	6	"	"	"	"
10	6	"	"	"	"
11	4	"	"	N	"
12	4	Exact	N	N	1

Each individual step in the evolution of our code was pretty straightforward; the transition from the sample in Example 3-6 to the exact answer in Example 3-7 is probably the most subtle. Along the way, as the code became faster and more useful, it also shrank in size. In the middle of the 19th century, Robert Browning observed that "less is more," and this table helps to quantify one instance of that minimalist philosophy.

We have seen three fundamentally different types of programs. Examples 3-2 and 3-3 are working Quicksorts, instrumented to count comparisons as they sort a real array. Examples 3-4 through 3-6 implement a simple model of Quicksort: they mimic one run of the algorithm, without actually doing the work of sorting. Examples 3-7 through 3-12 implement a more sophisticated model: they compute the true average number of comparisons without ever tracing any particular run.

Copyrighted material

The techniques used to achieve each program are summarized as follows:

- Examples 3-2, 3-4, 3-7: Fundamental change of problem definition.
- Examples 3-5, 3-6, 3-12: Slight change of function definition.
- Example 3-8: New data structure to implement dynamic programming.

These techniques are typical. We can often simplify a program by asking, "What problem do we really need to solve?" or, "Is there a better function to solve that problem?"

When I presented this analysis to undergraduates, the program finally shrank to zero lines of code and disappeared in a puff of mathematical smoke. We can reinterpret Example 3-7 as the following recurrence relation:

$$C_0 = 0 \qquad C_n = (n-1) + (1/n) \sum_{1 \le i \le n} C_{i-1} + C_{n-i}$$

This is precisely the approach taken by Hoare and later presented by D. E. Knuth in his classic *The Art of Computer Programming, Volume 3: Sorting and Searching* (Addison-Wesley). The programming tricks of re-expression and symmetry that give rise to Example 3-10 allow us to simplify the recursive part to:

$$C_n = n - 1 + (2/n) \sum_{0 \le i \le n-1} C_i$$

Knuth's technique to remove the summation sign gives (roughly) Example 3-11, which can be re-expressed as a system of two recurrence relations in two unknowns as:

$$C_0 = 0 \qquad S_0 = 0 \qquad S_n = S_{n-1} + 2C_{n-1} \qquad C_n = n - 1 + S_n/n$$

Knuth uses the mathematical technique of a "summing factor" to achieve the solution:

$$C_n = (n+1)(2H_{n+1} - 2) - 2n \sim 1.386 n \lg n$$

where H_n denotes the nth harmonic number, 1 + 1/2 + 1/3 + … 1/n. Thus we have smoothly progressed from experimenting on a program by augmenting it with probes to a completely mathematical analysis of its behavior.

With this formula, we end our quest. We have followed Einstein's famous advice to "make everything as simple as possible, but no simpler."

A Bonus Analysis

Goethe famously said that "architecture is frozen music." In exactly that sense, I assert that "data structures are frozen algorithms." And if we freeze the Quicksort algorithm, we get the data structure of a binary search tree. Knuth's publication presents that structure and analyzes its runtime with a recurrence relation similar to that for Quicksort.

Copyrighted material

If we wanted to analyze the average cost of inserting an element into a binary search tree, we could start with the code, augment it to count comparisons, and then conduct experiments on the data we gather. We could then simplify that code (and expand its power) in a manner very reminiscent of the previous section. A simpler solution is to define a new Quicksort that uses an *ideal partitioning* method that leaves the elements in the same relative order on both sides. That Quicksort is isomorphic to binary search trees, as illustrated in Figure 3-1.

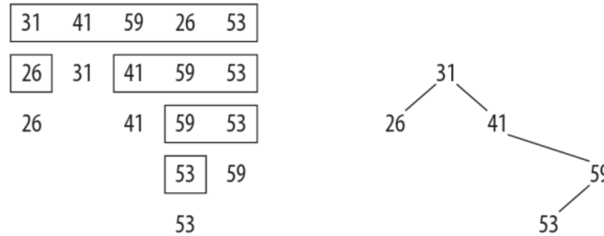


FIGURE 3-1. An ideal partitioning Quicksort and the corresponding binary search tree

The boxes on the left show an ideal-partitioning Quicksort in progress, and the graph on the right shows the corresponding binary search tree that has been built from the same input. Not only do the two processes make the same *number* of comparisons, they make exactly the same *set* of comparisons. Our previous analysis for the average performance of randomizing Quicksort on a set of distinct elements therefore gives us the average number of comparisons to insert randomly permuted distinct elements into a binary search tree.

What Is Writing?

In a weak sense, I "wrote" Examples 3-2 through 3-12 of the program. I wrote them first in scribbled notes, then on a chalkboard in front of undergraduates, and eventually in this chapter. I derived the programs systematically, I have spent considerable time analyzing them, and I believe that they are correct. Apart from the spreadsheet implementation of Example 3-11, though, I have never run any of the examples as a computer program.

In almost two decades at Bell Labs, I learned from many teachers (and especially from Brian Kernighan, whose chapter on the teaching of programming appears as Chapter 1 of this book) that "writing" a program to be displayed in public involves much more than typing symbols. One implements the program in code, runs it first on a few test cases, then builds thorough scaffolding, drivers, and a library of cases to beat on it systematically. Ideally, one mechanically includes the compiled source code into the text without human intervention. I wrote Example 3-1 (and all the code in *Programming Pearls*) in that strong sense.

As a point of honor, I wanted to keep my title honest by never implementing Examples 3-2 through 3-12. Almost four decades of computer programming have left me with deep

Copyrighted material

respect for the difficulty of the craft (well, more precisely, abject fear of bugs). I compromised by implementing Example 3-11 in a spreadsheet, and I tossed in an additional column that gave the closed-form solution. Imagine my delight (and relief) when the two matched exactly! And so I offer the world these beautiful unwritten programs, with some confidence in their correctness, yet painfully aware of the possibility of undiscovered error. I hope that the deep beauty I find in them will be unmarred by superficial blemishes.

In my discomfort at presenting these unwritten programs, I take consolation from the insight of Alan Perlis, who said, "Is it possible that software is not like anything else, that it is meant to be discarded: that the whole point is to see it as a soap bubble?"

Conclusion

Beauty has many sources. This chapter has concentrated on the beauty conferred by simplicity, elegance, and concision. The following aphorisms all express this overarching theme:

- Strive to add function by deleting code.
- A designer knows he has achieved perfection not when there is nothing left to add, but when there is nothing left to take away. (Saint-Exupéry)
- In software, the most beautiful code, the most beautiful functions, and the most beautiful programs are sometimes not there at all.
- Vigorous writing is concise. Omit needless words. (Strunk and White)
- The cheapest, fastest, and most reliable components of a computer system are those that aren't there. (Bell)
- Endeavor to do more and more with less and less.
- If I had more time, I would have written you a shorter letter. (Pascal)
- The Inventor's Paradox: The more ambitious plan may have more chance of success. (Pólya)
- Simplicity does not precede complexity, but follows it. (Perlis)
- Less is more. (Browning)
- Make everything as simple as possible, but no simpler. (Einstein)
- Software should sometimes be seen as a soap bubble. (Perlis)
- Seek beauty through simplicity.

Here endeth the lesson. Go thou and do likewise.

For those who desire more concrete hints, here are some ideas grouped into three main categories.

Copyrighted material

Analysis of programs

One way to gain insight into the behavior of a program is to instrument it and then run it on representative data, as in Example 3-2. Often, though, we are less concerned with the program as a whole than with individual aspects. In this case, for instance, we considered only the number of comparisons that Quicksort uses on the average and ignored many other aspects. Sedgewick ("The analysis of Quicksort programs," *Acta Informatica*, Vol. 7) studies issues such as the space it requires and many other components of runtime for a variety of Quicksort variants. By concentrating on the key issues, we can ignore (for a while) other aspects of the program. One of my articles, "A Case Study in Applied Algorithm Design" (*IEEE Computer*, Vol. 17, No. 2) describes how I once faced the problem of evaluating the performance of a *strip heuristic* for finding an approximate travelling salesman tour through `N` points in the unit square. I estimated that a complete program for the task might take 100 lines of code. After a series of steps similar in spirit to what we have seen in this chapter, I used a dozen-line simulation to give much more accuracy (and after completing my little simulation, I found that Beardwood et al. ["The Shortest Path Through Many Points," *Proc. Cambridge Philosophical Soc.*, Vol. 55] had re-expressed my simulation as a double integral, and thereby had solved the problem mathematically some two decades earlier).

Small pieces of code

I believe that computer programming is a practical skill, and I agree with Pólya that we "acquire any practical skill by imitation and practice." Programmers who long to write beautiful code should therefore read beautiful programs and imitate the techniques they learn as they write their own programs. I find that one of the most useful places to practice is on small code fragments, say of just one or two dozen lines. It was hard work but great fun preparing the second edition of *Programming Pearls*. I implemented every piece of code, and labored to pare each down to its essence. I hope that others enjoy reading the code as much as I enjoyed writing it.

Software systems

For specificity, I have described one tiny task in excruciating detail. I believe that the glory of these principles lies not in tiny code fragments, but rather in large programs and huge computer systems. Parnas ("Designing software for ease of extension and contraction," *IEEE T. Software Engineering*, Vol. 5, No. 2) offers techniques to whittle a system down to its essentials. For immediate applicability, don't forget the deep insight of Tom Duff: "Whenever possible, steal code."

Acknowledgments

I am grateful for the insightful comments of Dan Bentley, Brian Kernighan, Andy Oram, and David Weiss.

Copyrighted material

CHAPTER FOUR

Finding Things

Tim Bray

COMPUTERS CAN COMPUTE, BUT THAT'S NOT WHAT PEOPLE USE THEM FOR, MOSTLY. Mostly, computers store and retrieve information. *Retrieve* implies *find*, and in the time since the advent of the Web, search has become a dominant application for people using computers.

As data volumes continue to grow—both absolutely, and relative to the number of people or computers or anything, really—search becomes an increasingly large part of the life of the programmer as well. A few applications lack the need to locate the right morsel in some information store, but very few.

The subject of search is one of the largest in computer science, and thus I won't try to survey all of it or discuss the mechanics; in fact, I'll only consider one simple search technique in depth. Instead, I'll focus on the trade-offs that go into selecting search techniques, which can be subtle.

On Time

You really can't talk about search without talking about time. There are two different flavors of time that apply to problems of search. The first is the time it takes the search to run, which is experienced by the user who may well be staring at a message saying

Copyrighted material

something like "Loading...". The second is the time invested by the programmer who builds the search function, and by the programmer's management and customers waiting to use the program.

Problem: Weblog Data

Let's look at a sample problem to get a feel for how a search works in real life. I have a directory containing logfiles from my weblog (*http://www.tbray.org/ongoing*) from early 2003 to late 2006; as of the writing of this chapter, they recorded 140,070,104 transactions and occupied 28,489,788,532 bytes (uncompressed). All these statistics, properly searched, can answer lots of questions about my traffic and readership.

Let's look at a simple question first: which articles have been read the most? It may not be instantly obvious that this problem is about search, but it is. First of all, you have to search through the logfiles to find the lines that record someone fetching an article. Second, you have to search through those lines to find the name of the article they fetched. Third, you have to keep track, for each article, of how often it was fetched.

Here is an example of one line from one of these files, which wraps to fit the page in this book, but is a single long line in the file:

```
c80-216-32-218.cm-upc.chello.se - - [08/Oct/2006:06:37:48 -0700] "GET /ongoing/When/
200x/2006/10/08/Grief-Lessons HTTP/1.1" 200 5945 "http://www.tbray.org/ongoing/"
"Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322)
```

Reading from left to right, this tells us that:

Somebody from an organization named `chello` in Sweden,
who provided neither a username nor a password,
contacted my weblog early in the morning of October 8, 2006 (my server's time zone is seven hours off Greenwich),
and requested a resource named */ongoing/When/200x/2006/10/08/Grief-Lessons*
using the HTTP 1.1 protocol;
the request was successful and returned 5,945 bytes;
the visitor had been referred from my blog's home page,
and was using Internet Explorer 6 running on Windows XP.

This is an example of the kind of line I want: one that records the actual fetch of an article. There are lots of other lines that record fetching stylesheets, scripts, pictures, and so on, and attacks by malicious users. You can spot the kind of line I want by the fact that the article's name starts with `/ongoing/When/` and continues with elements for the decade, year, month, and day.

Our first step, then, should be to find lines that contain something like:

```
/ongoing/When/200x/2006/10/08/
```

Whatever language you're programming in, you could spend lots of time writing code to match this pattern character by character. Or you could apply regular expressions.

Copyrighted material

Regular Expressions

Regular expressions are special languages designed specifically for matching patterns in text. If you learn how to use them well, you'll save yourself immense amounts of time and irritation. I've never met a really accomplished programmer who wasn't a master of regular expressions (often called *regexps* for short). Chapter 1, by Brian Kernighan, is dedicated to the beauty of regular expressions.

Because the filenames on my web site match such a strict, date-based pattern, a very straightforward regular expression can find the logfile lines I'm interested in. Other sites' logfiles might require a more elaborate one. Here it is:

```
"GET /ongoing/When/\d\d\dx/\d\d\d\d/\d\d/\d\d/[^ .]+ "
```

A glance at this line instantly reveals one of the problems with regular expressions; they're not the world's most readable text. Some people might challenge their appearance in a book called *Beautiful Code*. Let's put that issue aside for a moment and look at this particular expression. The only thing you need to know is that in this particular flavor of regular expression:

`\d`
: Means "match any digit, 0 through 9"

`[^ .]`
: Means "match any character that's not a space or period"*

`+`
: Means "match one or more instances of whatever came just before the +"

That `[^ .]+`, then, means that the last slash has to be followed by a bunch of nonspace and nonperiod characters. There's a space *after* the + sign, so the regular expression stops when that space is found.

This regular expression won't match a line where the filename contains a period. So it will match `Grief-Lessons`, the example I showed earlier from my logfile, but not `IMG0038.jpg`.

Putting Regular Expressions to Work

A regular expression standing by itself, as shown above, can be used on the command line to search files. But it turns out that most modern computer languages allow you to use them directly in program code. Let's do that, and write a program that prints out only the lines that match the expression, which is to say a program that records all the times someone fetched an article from the weblog.

This example (and most other examples in this chapter) is in the Ruby programming language because I believe it to be, while far from perfect, the most readable of languages.

* People who have used regular expressions know that a period is a placeholder for "any character," but it's harder to remember that when a period is enclosed in square brackets, it loses the special meaning and refers to just a period.

Copyrighted material

If you don't know Ruby, learning it will probably make you a better programmer. In Chapter 29, the creator of Ruby, Yukihiro Matsumoto (generally known as "Matz"), discusses some of the design choices that have attracted me and so many other programmers to the language.

Example 4-1 shows our first Ruby program, with added line numbers on the left side. (All the examples in this chapter are available from the O'Reilly web site for this book.)

EXAMPLE 4-1. Printing article-fetch lines

```
1 ARGF.each_line do |line|
2   if line =~ %r{GET /ongoing/When/\d\d\dx/\d\d\d\d/\d\d/\d\d/[^ .]+ }
3     puts line
4   end
5 end
```

Running this program prints out a bunch of logfile lines that look like the first example. Let's have a line-by-line look at it:

Line 1

We want to read all the lines of the input, and we don't care whether they're from files named on the command line or are being piped in from another program on the standard input. The designers of Ruby believe strongly that programmers shouldn't have to write ugly code to deal with common situations, and this is a common situation. So, `ARGF` is a special variable that represents all the input sources. If the command line includes arguments, ARGF assumes they're names of files and opens them one by one; if there aren't any, it uses the standard input.

`each_line` is a method that you can call on pretty well any file-like object, such as `ARGF`. It reads the lines of input and passes them, one at a time, to a "block" of following code.

The following `do` says that the block getting the input stretches from there to the corresponding `end`, and the `|line|` asks that the `each_line` method load each line into the variable `line` before giving it to the block.

This kind of loop may surprise the eyes of a new convert to Ruby, but it's concise, powerful, and very easy to follow after just a bit of practice.

Line 2

This is a pretty straightforward `if` statement. The only magic is the `=~`, which means "matches" and expects to be followed by regular expression. You can tell Ruby that something is a regular expression by putting slashes before and after it—for example, `/this-is-a-regex/`. But the particular regular expression we want to use is full of slashes. So to use the slash syntax, you'd have to "escape" them by turning each `/` into `\/`, which would be ugly. In this case, therefore, the `%r` trick produces more beautiful code.

Line 3

We're inside the `if` block now. So, if the current `line` matches the regexp, the program executes `puts line`, which prints out the line and a line feed.

Copyrighted material

Lines 4 and 5

That's about all there is to it. The first `end` terminates the `if`, and the second terminates the `do`. They look kind of silly dangling off the bottom of the code, and the designers of Python have figured out a way to leave them out, which leads to some Python code being more beautiful than the corresponding Ruby.

So far, we've shown how regular expressions can be used to find the lines in the logfile that we're interested in. But what we're *really* interested in is counting the fetches for each article. The first step is to identify the article names. Example 4-2 is a slight variation on the previous program.

EXAMPLE 4-2. Printing article names

```
1 ARGF.each_line do |line|
2   if line =~ %r{GET /ongoing/When/\d\d\dx/(\d\d\d\d/\d\d/\d\d/[^ .]+) }
3     puts $1
4   end
5 end
```

The differences are subtle. In line 2, I've added a pair of parentheses (in boldface) around the interesting part of the article name in the regular expression. In line 3, instead of printing out the whole value of `line`, I print out `$1`, which in Ruby (and several other regular-expression-friendly languages) means "the first place in the regular expression marked with parentheses." You can mark lots of different pieces of the expression, and thus use `$2`, `$3`, and so on.

The first few lines of output produced by running this program over some logfile data look like this:

```
2003/10/10/FooCampMacs
2006/11/13/Rough-Mix
2003/05/22/StudentLookup
2003/11/13/FlyToYokohama
2003/07/31/PerlAngst
2003/05/21/RDFNet
2003/02/23/Democracy
2005/12/30/Spolsky-Recursion
2004/05/08/Torture
2004/04/27/RSSticker
```

Before we go to work determining the popularity of different articles, I'd like to argue that in some important ways, this code is beautiful. Take a moment and think of the code you'd have to write to look at an arbitrary chunk of text and do the same matching and selection work done by the parenthesized regexp. It would be quite a few lines of code, and it would be easy to get wrong. Furthermore, if the format of the logfile changed, fixing the pattern matcher would be error-prone and irritating.

Under the covers, the way that regular expressions work is also among the more wonderful things in computer science. It turns out that they can conveniently be translated into finite automata. These automata are mathematically elegant, and there are astoundingly

Copyrighted material

efficient algorithms for matching them against the text you're searching. The great thing is that when you're running an automaton, you have to look only once at each character in the text you're trying to match. The effect is that a well-built regular expression engine can do pattern matching and selection faster than almost any custom code, even if it were written in hand-optimized assembly language. That's beautiful.

I think that the Ruby code is pretty attractive, too. Nearly every character of the program is doing useful work. Note that there are no semicolons on the ends of the lines, nor parentheses around the conditional block, and that you can write `puts line` instead of `puts(line)`. Also, variables aren't declared—they're just used. This kind of stripped-down language design makes for programs that are shorter and easier to write, as well as (more important) easier to read and easier to understand.

Thinking in terms of time, regular expressions are a win/win. It takes the programmer way less time to write them than the equivalent code, it takes less time to deliver the program to the people waiting for it, it uses the computer really efficiently, and the program's user spends less time sitting there bored.

Content-Addressable Storage

Now we're approaching the core of our problem, computing the popularity of articles. We'll have to pull the article name out of each line, look it up to see how many times it's been fetched, add one to that number, and then store it away again.

This may be the most basic of search patterns: we start with a *key* (what we're using to search—in this case, an article name), and we're looking for a *value* (what we want to find—in this case, the number of times the article has been fetched). Here are some other examples:

Key	Value
Word	List of web pages containing the word
Employee number	Employee's personnel record
Passport number	"true" or "false," indicating whether the person with that passport should be subject to extra scrutiny

What programmers really want in this situation is a very old idea in computer science: *content-addressable memory*, also known as an *associative store* and various other permutations of those words. The idea is to put the key in and get the value out. There actually exists hardware which does just that; it mostly lives deep in the bowels of microprocessors, providing rapid access to page tables and memory caches.

The good news is that you, the programmer, using any modern computer language, have access to excellent software implementations of associative memory. Different languages call these implementations different things. Often they are implemented as hash tables; in

Copyrighted material

Java, Perl, and Ruby, which use this technique, they are called *Hashes*, *HashMaps*, or something similar. In Python, they are called *dictionaries*, and in the computer algebra language Maple, simply *tables*.

Now if you're an eager search-algorithm fan just itching to write your own super-efficient search, this may sound like bad news, not good news. But think about those flavors of time; if you use the built-in associative store, the amount of programmer time and management invested in writing search algorithms goes to nearly zero.

By writing your own search, you *might* be able to save a little computer (and thus end-user) time, compared to the built-in version, but on the other hand, you might not; the people who write these things tend to be pretty clever. Andrew Kuchling has written Chapter 18 of this book on one such effort.

Associative stores are so important that dynamically typed languages such as Ruby and Python have not only built-in support, but special syntax for defining and using them. Let's use Ruby's hashes to count article popularity in Example 4-3.

EXAMPLE 4-3. Counting article fetches

```
1 counts = {}
2 counts.default = 0
3
4 ARGF.each_line do |line|
5   if line =~ %r{GET /ongoing/When/\d\d\dx/(\d\d\d\d/\d\d/\d\d/[^ .]+) }
6     counts[$1] += 1
7   end
8 end
```

This program isn't that much different from the version in Example 4-2. Line 1 creates an empty Hash called `counts`. Line 2 gives the array a "default value" of zero; hold on for an explanation of that.

Then, in line 6, instead of printing out the article name, the name serves as the key to look up the number of fetches of this article seen so far in `counts`, add one to it, and store the value.

Now, consider what happens when the program sees some article name stored in `$1` for the first time. I could write code along the lines of "if there is a `counts[$1]`, then add one to it; otherwise, set `counts[$1]` to one." The designers of Ruby hate that kind of awkwardness; this is why they provided the notion of a "default value" for a Hash. If you look up a key the Hash doesn't know about, it says "OK, zero," allowing you to write `counts[$1] += 1` and have it always just work.

I originally stated the problem as "Which of my articles have been read the most?" That's kind of fuzzy; let's interpret it to mean "Print out the top 10 most popular articles." The resulting program is shown in Example 4-4.

Copyrighted material

EXAMPLE 4-4. Reporting the most popular articles

```
1 counts = {}
2 counts.default = 0
3
4 ARGF.each_line do |line|
5   if line =~ %r{GET /ongoing/When/\d\d\dx/(\d\d\d\d/\d\d/\d\d/[^ .]+) }
6     counts[$1] += 1
7   end
8 end
9
10 keys_by_count = counts.keys.sort { |a, b| counts[b] <=> counts[a] }
11 keys_by_count[0 .. 9].each do |key|
12   puts "#{counts[key]}: #{key}"
13 end
```

Line 10 looks a little less beautiful to me than most Ruby code, but it's easy enough to understand. The `keys` method of `counts` returns an array containing all of the Hash's keys. Because of the hash implementation, the keys are stored in no predictable order, and are also returned by the `keys` method in random order. So, I have to sort them and store them back in a new array.

In Ruby, `sort` is accompanied by a code block, here enclosed in curly braces. (In Ruby, you can delimit a block either with `do` and `end` or with { and }.) The sort works its way back and forth through the array being sorted, passing pairs of elements to the block, which has to return a negative number, 0, or a positive number depending on whether the first element is less than, equal to, or greater than the second.

In this case, we want to get the data out of the hash in an order defined by the values (the counts themselves) rather than by the filenames (the keys), so we have to sort the keys by their values. Have a close look at the code, and you'll see how it works. Because this is something that people do all the time, I'm surprised that Ruby's Hash doesn't come with `sort_by_value`.

We use a decreasing order for the sort so that, no matter how many articles we've found, we know the first 10 items in `keys_by_count` represent the top 10 articles in popularity.

Now that we have an array of keys (article names) sorted in descending order of how many times they've been fetched, we can accomplish our assignment by printing out the first 10. Line 11 is simple, but a word is in order about that `each` method. In Ruby, you almost never see a `for` statement because anything whose elements you might want to loop through has an `each` method that does it for you.

Line 12 may be a little hard to read for the non-Rubyist because of the `#{}` syntax, but it's pretty straightforward.

So, let's declare victory on our first assignment. It took us only 13 lines of easy-to-read code. A seasoned Rubyist would have squeezed the last three lines into one.

Let's run this thing and see what it reports. Instead of running it over the whole 28 GB, let's just use it on a week's data: a mere 1.2 million records comprising 245 MB.

Copyrighted material

```
~/dev/bc/ 548> zcat ~/ongoing/logs/2006-12-17.log.gz | \
               time ruby code/report-counts.rb
4765: 2006/12/11/Mac-Crash
3138: 2006/01/31/Data-Protection
1865: 2006/12/10/EMail
1650: 2006/03/30/Teacup
1645: 2006/12/11/Java
1100: 2006/07/28/Open-Data
900: 2006/11/27/Choose-Relax
705: 2003/09/18/NXML
692: 2006/07/03/July-1-Fireworks
673: 2006/12/13/Blog-PR
       13.54 real         7.49 user         0.73 sys
```

This run took place on my 1.67 GHz Apple PowerBook. The results are unsurprising, but the program does seem kind of slow. Should we worry about performance?

Time to Optimize?

I was wondering whether my sample run was really unreasonably slow, so I pulled together a very similar program in Perl, a language that is less beautiful than Ruby but is *extremely* fast. Sure enough, the Perl version took half the time. So, should we try to optimize?

We need to think about time again. Yes, we might be able to make this run faster, and thus reduce the program execution time and the time a user spends waiting for it, but to do this we'd have to burn some of the programmer's time, and thus the time the user waits for the programmer to get the program written. In most cases, my instinct would be that 13.54 seconds to process a week's data is OK, so I'd declare victory. But let's suppose we're starting to get gripes from people who use the program, and we'd like to make it run faster.

Glancing over Example 4-4, we can see that the program falls into two distinct parts. First, it reads all the lines and tabulates the fetches; then it sorts them to find the top 10.

There's an obvious optimization opportunity here: why bother sorting all the fetch tallies when all we really want to do is pick the top 10? It's easy enough to write a little code to run through the array once and pick the 10 highest elements.

Would that help? I found out by instrumenting the program to find out how much time it spent doing its two tasks. The answer was (averaging over a few runs) 7.36 seconds in the first part and 0.07 in the second. Which is to say, "No, it wouldn't help."

Might it be worthwhile to try to optimize the first part? Probably not; all it does is match regular expressions, and store and retrieve data using a Hash, and these are among the most heavily optimized parts of Ruby.

So, getting fancy in replacing that sort would probably waste the time of the programmer and the customer waiting for the code, without saving any noticeable amount of computer or waiting-user time. Also, experience would teach that you're not apt to go much faster than Perl does for this kind of task, so the amount of speedup you're going to get is pretty well bounded.

Copyrighted material

We've just finished writing a program that does something useful and turns out to be all about search. But we haven't come anywhere near actually writing any search algorithms. So, let's do that.

SOME HISTORY OF TALLYING

In the spirit of credit where credit is due, the notion of getting real work done by scanning lines of textual input using regular expressions and using a content-addressable store to build up results was first popularized in the *awk* programming language, whose name reflects the surnames of its inventors Aho, Weinberger, and Kernighan.

This work, of course, was based on the then-radical Unix philosophy—due mostly to Ritchie and Thompson—that data should generally be stored in files in lines of text, and to some extent validated the philosophy.

Larry Wall took the ideas behind *awk* and, as the author of Perl, turned them into a high-performance, industrial-strength, general-purpose tool that doesn't get the credit it deserves. It served as the glue that has held together the world's Unix systems, and subsequently large parts of the first-generation Web.

Problem: Who Fetched What, When?

Running a couple of quick scripts over the logfile data reveals that there are 12,600,064 instances of an article fetch coming from 2,345,571 different hosts. Suppose we are interested in who was fetching what, and when? An auditor, a police officer, or a marketing professional might be interested.

So, here's the problem: given a hostname, report what articles were fetched from that host, and when. The result is a list; if the list is empty, no articles were fetched.

We've already seen that a language's built-in hash or equivalent data structure gives the programmer a quick and easy way to store and look up key/value pairs. So, you might ask, why not use it?

That's an excellent question, and we should give the idea a try. There are reasons to worry that it might not work very well, so in the back of our minds, we should be thinking of a Plan B. As you may recall if you've ever studied hash tables, in order to go fast, they need to have a small load factor; in other words, they need to be mostly empty. However, a hash table that holds 2.35 million entries and is still mostly empty is going to require the use of a whole lot of memory.

Copyrighted material

To simplify things, I wrote a program that ran over all the logfiles and pulled out all the article fetches into a simple file; each line has the hostname, the time of the transaction, and the article name. Here are the first few lines:

```
crawl-66-249-72-77.googlebot.com 1166406026 2003/04/08/Riffs
egspd42470.ask.com 1166406027 2006/05/03/MARS-T-Shirt
84.7.249.205 1166406040 2003/03/27/Scanner
```

(The second field, the 10-digit number, is the standard Unix/Linux representation of time as the number of seconds since the beginning of 1970.)

Then I wrote a simple program to read this file and load a great big hash. Example 4-5 shows the program.

EXAMPLE 4-5. Loading a big hash

```
1 class BigHash
2
3   def initialize(file)
4     @hash = {}
5     lines = 0
6     File.open(file).each_line do |line|
7       s = line.split
8       article = s[2].intern
9       if @hash[s[0]]
10        @hash[s[0]] << [ s[1], article ]
11      else
12        @hash[s[0]] = [ s[1], article ]
13      end
14      lines += 1
15      STDERR.puts "Line: #{lines}" if (lines % 100000) == 0
16    end
17  end
18
19  def find(key)
20    @hash[key]
21  end
22
23 end
```

The program should be fairly self-explanatory, but line 15 is worth a note. When you're running a big program that's going to take a lot of time, it's very disturbing when it works away silently, maybe for hours. What if something's wrong? What if it's going incredibly slow and will never finish? So, line 15 prints out a progress report after every 100,000 lines of input, which is reassuring.

Running this program was interesting. It took about 55 minutes of CPU time to load up the hash, and the program grew to occupy 1.56 GB of memory. A little calculation suggests that it costs around 680 bytes to store the information for each host, or slicing the data another way, about 126 bytes per fetch. This is a little scary, but probably reasonable for a hash table.

Copyrighted material

Retrieval performance was excellent. I ran 2,000 queries, half of which were randomly selected hosts from the log and thus succeeded, while the other half were those same hostnames reversed, none of which succeeded. The 2,000 queries completed in an average of about .02 seconds, so Ruby's hash implementation can look up records in a hash containing 12 million or so records thousands of times per second.

Those 55 minutes to load up the data are troubling, but there are some tricks to address that. You could, for example, load it up once, then serialize the hash out and read it back in. And I didn't try particularly hard to optimize the program.

The program was easy and quick to write, and it runs fast once it's initialized, so its performance is good both in terms of waiting-for-the-program time and waiting-for-the-programmer time. Still, I'm unsatisfied. I have the feeling that there ought to be a way to get this kind of performance while burning less memory, less startup time, or both. It involves writing our own search code, though.

Binary Search

Nobody gets a Computer Science degree without studying a wide variety of search algorithms: trees, heaps, hashes, lists, and more. My favorite among all these is binary search. Let's try it on the who-fetched-what-when problem and then look at what makes it beautiful.

My first attempt at putting binary search to use was quite disappointing; while the data took 10 minutes less to load, it required almost 100 MB more memory than with the hash. Clearly, there are some surprising things about the Ruby array implementation. The search also ran several times slower (but still in the range of thousands per second), but this is not surprising at all because the algorithm is running in Ruby code rather than with the underlying hardcoded hash implementation.

The problem is that in Ruby everything is an object, and arrays are fairly abstracted things with lots of built-in magic. So, let's reimplement the program in Java, in which integers are just integers, and arrays come with very few extras.*

Nothing could be simpler, conceptually, than binary search. You divide your search space in two and see whether you should be looking in the top or bottom half; then you repeat the exercise until done. Instructively, there are a great many ways to code this algorithm incorrectly, and several widely published versions contain bugs. The implementation mentioned in "On the Goodness of Binary Search," and shown in Java in Example 4-6, is based on one I learned from Gaston Gonnet, the lead developer of the Maple language for symbolic mathematics and currently Professor of Computer Science at ETH in Zürich.

* This discussion of binary search borrows heavily from my 2003 piece, "On the Goodness of Binary Search," available online at *http://www.tbray.org/ongoing/When/200x/2003/03/22/Binary*.

Copyrighted material

EXAMPLE 4-6. Binary search

```
1 package binary;
2
3  public class Finder {
4    public static int find(String[] keys, String target) {
5      int high = keys.length;
6      int low = -1;
7      while (high - low > 1) {
8        int probe = (low + high) >>> 1;
9        if (keys[probe].compareTo(target) > 0)
10         high = probe;
11       else
12         low = probe;
13     }
14     if (low == -1 || keys[low].compareTo(target) != 0)
15       return -1;
16     else
17       return low;
18   }
19 }
```

Key aspects of this program are as follows:

- In lines 5–6, note that the `high` and `low` bounds are set one off the ends of the array, so neither are initially valid indices. This eliminates all sorts of corner cases.
- The loop that starts in line 7 runs until the high and low bounds are adjacent; there is no testing to see whether the target has been found. Think for a minute whether you agree with this choice; we'll return to the question later.

  The loop has two invariants. `low` is either –1 or points to something less than or equal to the target value. `high` is either one off the top of the array or points to something strictly greater than the target value.
- Line 8 is particularly interesting. In an earlier version it read:

  ```
  probe = (high + low) / 2;
  ```

  but in June 2006, Java guru Josh Bloch showed how, in certain obscure circumstances, that code could lead to integer overflow (see *http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html*). It is sobering indeed that, many decades into the lifetime of computer science, we are still finding bugs in our core algorithms. (The issue is also discussed by Alberto Savoia in Chapter 7.)

  At this point, Rubyists will point out that modern dynamic languages such as Ruby and Python take care of integer overflow for you, and thus don't have this bug.
- Because of the loop invariant, once I'm done with the loop, I just need to check `low` (lines 14–17). If it's not –1, either it points to something that matches the target, or the target isn't there.

The Java version took only six and a half minutes to load, and it ran successfully, using less than 1 GB of heap. Also, while it's harder to measure CPU time in Java than in Ruby, there was no perceptible delay in running the same 2,000 searches.

Copyrighted material

Binary Search Trade-offs

Binary search has some very large advantages. First of all, its performance is $O(\log_2 N)$. People often don't really grasp how powerful this is. On a 32-bit computer, the biggest $\log_2$ you'll ever encounter is 32 (similarly, 64 on a 64-bit computer), and any algorithm that competes in an upper bound of a few dozen steps will be "good enough" for many real-world scenarios.

Second, the binary-search code is short and simple. Code that is short and simple is beautiful, for a bunch of reasons. Maybe the most important is that it's easier to understand, and understanding code is harder than writing it. There are fewer places for bugs to hide. Also, compact code plays better with instruction sets, I-caches, and JIT compilers, and thus tends to run faster.

Third, once you've got that sorted array, you don't need any more index structures; binary search is very space-efficient.

The big downside to binary search is that the data has to be kept in order in memory. There are some data sets for which this is impossible, but fewer than you might think. If you think you have too much data to fit in memory, check the price of RAM these days and make sure. Any search strategy that requires going to disk is going to be immensely more complex, and in many scenarios slower.

Suppose you need to update the data set; you might think that would rule out binary search because you have to update a huge, contiguous array in memory. But that turns out to be easier than you might think. In fact, your program's memory is scattered randomly all over the computer's physical RAM, with the operating system's paging software making it look sequential; you can do the same kind of trick with your own data.

Some might argue that since a hash table is $O(1)$, that has to be better than binary search's $O(\log_2 N)$. In practice, the difference may not be that significant; set up an experiment sometime and do some measurements. Also, consider that hash tables, with the necessary collision-resolution code, are considerably more complex to implement.

I don't want to be dogmatic, but in recent years, I've started to take the following approach to search problems:

1. Try to solve it using your language's built-in hash tables.
2. Then try to solve it with binary search.
3. Only then should you reluctantly start to consider other more complex options.

Escaping the Loop

Some look at my binary-search algorithm and ask why the loop always runs to the end without checking whether it's found the target. In fact, this is the correct behavior; the math is beyond the scope of this chapter, but with a little work, you should be able to get an intuitive feeling for it—and this is the kind of intuition I've observed in some of the great programmers I've worked with.

Copyrighted material

Let's think about the progress of the loop. Suppose you have n elements in the array, where n is some really large number. The chance of finding the target the first time through is $1/n$, a really small number. The next iteration (after you divide the search set in half) is $1/(n/2)$—still small—and so on. In fact, the chance of hitting the target becomes significant only when you're down to 10 or 20 elements, which is to say maybe the last four times through the loop. And in the case where the search fails (which is common in many applications), those extra tests are pure overhead.

You could do the math to figure out when the probability of hitting the target approaches 50 percent, but qualitatively, ask yourself: does it make sense to add extra complexity to each step of an $O(\log_2 N)$ algorithm when the chances are it will save only a small number of steps at the end?

The take-away lesson is that binary search, done properly, is a two-step process. First, write an efficient loop that positions your `low` and `high` bounds properly, then add a simple check to see whether you hit or missed.

Search in the Large

When most people think of search they think of web search, as offered by Yahoo!, Google, and their competitors. While ubiquitous web search is a new thing, the discipline of full-text search upon which it is based is not. Most of the seminal papers were written by Gerald Salton at Cornell as far back as the early 1960s. The basic techniques for indexing and searching large volumes of text have not changed dramatically since then. What *has* changed is how result ranking is done.*

Searching with Postings

The standard approach to full-text search is based on the notion of a *posting*, which is a small, fixed-size record. To build an index, you read all the documents and, for each word, create a posting that says word *x* appears in document *y* at position *z*. Then you sort all the words together, so that for each unique word you have a list of postings, each a pair of numbers consisting of a document ID and the text's offset in that document.

Because postings are small and fixed in size, and because you tend to have a huge number of them, a natural approach is to use binary search. I have no idea of the details of how Google or Yahoo! do things, but I'd be really unsurprised to hear that those tens of thousands of computers spend a whole lot of their time binary-searching big arrays of postings.

People who are knowledgeable about search shared a collective snicker a few years ago when the number of documents Google advertised as searching, after having been stuck at two billion and change for some years, suddenly became much larger and then kept

* This discussion of full-text search borrows heavily from my 2003 series, *On Search*, available online at *http://www.tbray.org/ongoing/When/200x/2003/07/30/OnSearchTOC*. The series covers the topic of search quite broadly, including issues of user experience, quality control, natural language processing, intelligence, internationalization, and so on.

Copyrighted material

growing. Presumably they had switched the document ID in all those postings from 32-bit to 64-bit numbers.

Ranking Results

Given a word, searching a list of postings to figure out which documents contain it is not rocket science. A little thought shows that combining the lists to do AND and OR queries and phrase search is also simple, conceptually at least. What's hard is sorting the result list so that the good results show up near the top. Computer science has a subdiscipline called Information Retrieval (IR for short) that focuses almost entirely on this problem. Historically, the results had been very poor, up until recently.

Searching the Web

Google and its competitors have been able to produce good results in the face of unimaginably huge data sets and populations of users. When I say "good," I mean that high-quality results appear near the top of the result list, and that the result list appears quickly.

The promotion of high-quality results is a result of many factors, the most notable of which is what Google calls *PageRank*, based largely on link counting: pages with lots of hyperlinks pointing at them are deemed to be more popular and thus, by popular vote, winners.

In practice, this seems to work well. A couple of interesting observations follow. First, until the rise of PageRank, the leaders in the search-engine space were offerings such as Yahoo! and DMoz, which worked by categorizing results; so, the evidence seems to suggest that it's more useful to know how popular something is than to know what it's about.

Second, PageRank is applicable only to document collections that are richly populated with links back and forth between the documents. At the moment, two document collections qualify: the World Wide Web and the corpus of peer-reviewed academic publications (which have applied PageRank-like methods for decades).

The ability of large search engines to scale up with the size of data and number of users has been impressive. It is based on the massive application of parallelism: attacking big problems with large numbers of small computers, rather than a few big ones. One of the nice things about postings is that each posting is independent of all the others, so they naturally lend themselves to parallel approaches.

For example, an index based on doing binary search in arrays of postings is fairly straightforward to partition. In an index containing only English words, you could easily create 26 partitions (the term used in the industry is *shards*), one for words beginning with each letter. Then you can make as many copies as you need of each shard. Then, a huge volume of word-search queries can be farmed out across an arbitrarily large collection of cooperating search nodes.

Copyrighted material

This leaves the problem of combining search results for multiword or phrase searches, and this requires some real innovation, but it's easy to see how the basic word-search function could be parallelized.

This discussion is a little unfair in that it glosses over a huge number of important issues, notably including fighting the Internet miscreants who continually try to outsmart search-engine algorithms for commercial gain.

Conclusion

It is hard to imagine any computer application that does not involve storing data and finding it based on its content. The world's single most popular computer application, web search, is a notable example.

This chapter has considered some of the issues, notably bypassing the traditional "database" domain and the world of search strategies that involve external storage. Whether operating at the level of a single line of text or billions of web documents, search is central. From the programmer's point of view, it also needs to be said that implementing searches of one kind or another is, among other things, fun.

Copyrighted material

Copyrighted material

CHAPTER FIVE

Correct, Beautiful, Fast (in That Order): Lessons from Designing XML Verifiers

Elliotte Rusty Harold

THIS IS THE STORY OF TWO ROUTINES THAT PERFORM INPUT VERIFICATION FOR XML, the first in JDOM, and the second in XOM. I was intimately involved in the development of both, and while the two code bases are completely separate and share no common code, the ideas from the first clearly trickled into the second. The code, in my opinion, gradually became more beautiful. It certainly became faster.

Speed was the driving factor in each successive refinement, but in this case the improvements in speed were accompanied by improvements in beauty as well. I hope to dispel the myth that fast code must be illegible, ugly code. On the contrary, I believe that more often than not, improvements in beauty *lead to* improvements in execution speed, especially taking into account the impact of modern optimizing compilers, just-in-time compilers, RISC (reduced instruction set computer) architectures, and multi-core CPUs.

The Role of XML Validation

XML achieves interoperability by rigorously enforcing certain rules about what may and may not appear in an XML document. With a few very small exceptions, a conforming processor can process any well-formed XML document and can identify (and not attempt to process) malformed documents. This ensures a high degree of interoperability between

Copyrighted material

platforms, parsers, and programming languages. You don't have to worry that your parser won't read my document because yours was written in C and runs on Unix, while mine was written in Java and runs on Windows.

Fully maintaining XML correctness normally involves two redundant checks on the data:

1. Validation occurs on input. As a parser reads an XML document, it checks the document for well-formedness and, optionally, validity. *Well-formedness* checks purely syntactic constraints, such as whether every start tag has a matching end tag. This is required of all XML parsers. *Validity* means that only elements and attributes specifically listed in a Document Type Definition (DTD) appear, and only in the proper positions.
2. Verification happens on output. When generating an XML document through an XML API such as DOM, JDOM, or XOM, the parser checks all strings passing through the API to make sure they're legal in XML.

While input validation is more thoroughly defined by the XML specification, output verification can be equally important. In particular, it is critical for debugging and making sure that the code is correct.

The Problem

The very first beta releases of JDOM did not verify the strings used to create element names, text content, or pretty much anything else. Programs were free to generate element names that contained whitespace, comments that ended in hyphens, text nodes that contained nulls, and other malformed content. Maintaining the correctness of the generated XML was completely left up to the client programmer.

This bothered me. While XML is simpler than some alternatives, it is not simple enough that it can be fully understood without immersing yourself in specification arcana, such as exactly which Unicode code points are or are not legal in XML names and text content.

JDOM aimed to be an API that brought XML to the masses. JDOM aimed to be an API that, unlike DOM, did not require a two-week course and an expensive expert mentor to learn to use properly. To enable this, JDOM needed to lift as much of the burden of understanding XML from the programmer as possible. Properly implemented, JDOM would keep the programmer from making mistakes.

There are numerous ways JDOM could do this. Some of them fell out as a direct result of its data model. For instance, in JDOM it is not possible to overlap elements (`<p>Sally said, <quote>let's go the park.</p>. Then let's play ball.</quote>`). Because JDOM's internal representation is a tree, there's simply no way to generate this markup from JDOM. However, a number of other constraints need to be checked explicitly, such as whether:

- The name of an element, attribute, or processing instruction is a legal XML name
- Local names do not contain colons

Copyrighted material

- Attribute namespaces do not conflict with the namespaces of their parent element or sibling attributes
- Every Unicode surrogate character appears as part of a surrogate pair consisting of one high surrogate followed by one low surrogate
- Processing instruction data does not contain the two-character string ?>

Whenever the client supplies a string for use in one of these areas, it should be checked to see that it meets the relevant constraints. The details vary, but the basic approach is the same.

For purposes of this chapter, I'm going to examine the rules for checking XML 1.0 element names.

In the XML 1.0 specification (part of which is given in Example 5-1), rules are given in a Backus-Naur Form (BNF) grammar. Here `#x`*dddd* represents the Unicode code point with the hexadecimal value *dddd*. `[#x`*dddd*`-#x`*eeee*`]` represents all Unicode code points from `#x`*dddd* to `#x`*eeee*.

EXAMPLE 5-1. BNF grammar for checking XML names (abridged)

```
BaseChar    ::= [#x0041-#x005A] | [#x0061-#x007A] | [#x00C0-#x00D6]
NameChar    ::= Letter | Digit | '.' | '-' | '_' | ':' | CombiningChar | Extender
Name        ::= (Letter | '_' | ':') (NameChar)*
Letter      ::= BaseChar | Ideographic
Ideographic ::= [#x4E00-#x9FA5] | #x3007 | [#x3021-#x3029]
Digit       ::= [#x0030-#x0039] | [#x0660-#x0669] | [#x06F0-#x06F9]
              | [#x0966-#x096F] | [#x09E6-#x09EF] | [#x0A66-#x0A6F]
              | [#x0AE6-#x0AEF] | [#x0B66-#x0B6F] | [#x0BE7-#x0BEF]
              | [#x0C66-#x0C6F] | [#x0CE6-#x0CEF] | [#x0D66-#x0D6F]
              | [#x0E50-#x0E59] | [#x0ED0-#x0ED9] | [#x0F20-#x0F29]
Extender    ::=  #x00B7 | #x02D0 | #x02D1 | #x0387 | #x0640 | #x0E46 | #x0EC6
               | #x3005 | [#x3031-#x3035] | [#x309D-#x309E] | [#x30FC-#x30FE]
              | [#x00D8-#x00F6] | [#x00F8-#x00FF] | [#x0100-#x0131]
              | [#x0134-#x013E] | [#x0141-#x0148] | [#x014A-#x017E]
              | [#x0180-#x01C3] ...
CombiningChar ::=  [#x0300-#x0345] | [#x0360-#x0361] | [#x0483-#x0486]
                 | [#x0591-#x05A1] | [#x05A3-#x05B9] | [#x05BB-#x05BD] | #x05BF
                 | [#x05C1-#x05C2] | #x05C4 | [#x064B-#x0652] | #x0670
                 | [#x06D6-#x06DC] | [#x06DD-#x06DF] | [#x06E0-#x06E4]
                 | [#x06E7-#x06E8] | [#x06EA-#x06ED]...
```

The complete set of rules would take up several pages here, as there are over 90,000 characters in Unicode to consider. In particular, the rules for `BaseChar` and `CombiningChar` have been shortened in this example.

To verify that a string is a legal XML name, it is necessary to iterate through each character in the string and verify that it is a legal name character as defined by the `NameChar` production.

Copyrighted material

Version 1: The Naïve Implementation

My initial contribution to JDOM (shown in Example 5-2) simply deferred the rule checks to Java's `Character` class. The `checkXMLName` method returns an error message if an XML name is invalid, and null if it's valid. This itself is a questionable design; it should probably throw an exception if the name is invalid, and return void in all other cases. Later in this chapter, you'll see how future versions addressed this.

EXAMPLE 5-2. The first version of name character verification

```
private static String checkXMLName(String name) {
    // Cannot be empty or null
    if ((name == null) || (name.length() == 0) || (name.trim().equals(""))) {
        return "XML names cannot be null or empty";
    }

    // Cannot start with a number
    char first = name.charAt(0);
    if (Character.isDigit(first)) {
        return "XML names cannot begin with a number.";
    }
    // Cannot start with a $
    if (first == '$') {
        return "XML names cannot begin with a dollar sign ($).";
    }
    // Cannot start with a _
    if (first == '-') {
        return "XML names cannot begin with a hyphen (-).";
    }

    // Ensure valid content
    for (int i=0, len = name.length(); i<len; i++) {
        char c = name.charAt(i);
        if ((!Character.isLetterOrDigit(c))
            && (c != '-')
            && (c != '$')
            && (c != '_')) {
            return c + " is not allowed in XML names.";
        }
    }

    // We got here, so everything is OK
    return null;
}
```

This method was straightforward and easy to understand. Unfortunately, it was wrong. In particular:

- It allowed names that contained colons. Because JDOM attempted to maintain namespace well-formedness, this had to be fixed.
- The Java `Character.isLetterOrDigit` and `Character.isDigit` methods aren't perfectly aligned with XML's definition of letters and digits. Java considers some characters as letters that XML doesn't, and vice versa.
- The Java rules change from one version of Java to the next. XML's rules don't.

Copyrighted material

Nonetheless, this was a reasonable first attempt. It did catch a large percentage of malformed names and didn't reject too many well-formed ones. It worked especially well in the common case when all the names were ASCII. Even so, JDOM strived for broader applicability than that. An improved implementation that actually followed XML's rules was called for.

Version 2: Imitating the BNF Grammar O(N)

My next contribution to JDOM manually translated the BNF productions into a series of `if-else` statements. The result looked like Example 5-3. You'll notice that this version is quite a bit more complicated.

EXAMPLE 5-3. BNF-based name character verification

```
private static String checkXMLName(String name) {
    // Cannot be empty or null
    if ((name == null) || (name.length( ) == 0)
                       || (name.trim( ).equals(""))) {
        return "XML names cannot be null or empty";
    }


    // Cannot start with a number
    char first = name.charAt(0);
    if (!isXMLNameStartCharacter(first)) {
        return "XML names cannot begin with the character \"" +
               first + "\"";
    }
    // Ensure valid content
    for (int i=0, len = name.length( ); i<len; i++) {
        char c = name.charAt(i);
        if (!isXMLNameCharacter(c)) {
            return "XML names cannot contain the character \"" + c + "\"";
        }
    }

    // We got here, so everything is OK
    return null;
}

public static boolean isXMLNameCharacter(char c) {

  return (isXMLLetter(c) || isXMLDigit(c) || c == '.' || c == '-'
                         || c == '_' || c == ':' || isXMLCombiningChar(c)
                         || isXMLExtender(c));

}

public static boolean isXMLNameStartCharacter(char c) {
  return (isXMLLetter(c) || c == '_' || c ==':');
}
```


Copyrighted material

Instead of simply reusing Java's `Character.isLetterOrDigit` and `Character.isDigit` methods, the `checkXMLName` method in Example 5-3 delegates the checks to `isXMLNameCharacter` and `isXMLNameStartCharacter`. These methods further delegate to methods matching the other BNF productions for the different types of characters: letters, digits, combining characters, and extenders. Example 5-4 shows one of these methods, `isXMLDigit`. Notice that this method considers not only the ASCII digits, but also the other digit characters included in Unicode 2.0. The `isXMLLetter`, `isXMLCombiningChar`, and `isXMLExtender` methods follow the same pattern. They're just longer.

EXAMPLE 5-4. XML-based digit character verification

```
public static boolean isXMLDigit(char c) {

    if (c >= 0x0030 && c <= 0x0039) return true;
    if (c >= 0x0660 && c <= 0x0669) return true;
    if (c >= 0x06F0 && c <= 0x06F9) return true;
    if (c >= 0x0966 && c <= 0x096F) return true;

    if (c >= 0x09E6 && c <= 0x09EF) return true;
    if (c >= 0x0A66 && c <= 0x0A6F) return true;
    if (c >= 0x0AE6 && c <= 0x0AEF) return true;

    if (c >= 0x0B66 && c <= 0x0B6F) return true;
    if (c >= 0x0BE7 && c <= 0x0BEF) return true;
    if (c >= 0x0C66 && c <= 0x0C6F) return true;

    if (c >= 0x0CE6 && c <= 0x0CEF) return true;
    if (c >= 0x0D66 && c <= 0x0D6F) return true;
    if (c >= 0x0E50 && c <= 0x0E59) return true;

    if (c >= 0x0ED0 && c <= 0x0ED9) return true;
    if (c >= 0x0F20 && c <= 0x0F29) return true;

    return false;

}
```

This approach satisfied the basic goals of the upgrade. It worked, and its operation was obvious. There was a clear mapping from the XML specification to the code. We could declare victory and go home.

Well, not quite. This was where the ugly specter of performance raised its head.

Version 3: First Optimization O(log N)

As Donald Knuth once said, "Premature optimization is the root of all evil in programming." However, although optimization matters less often than programmers think, it does matter; and this was one of the minority cases where it matters.

Profiling proved that JDOM was spending a significant chunk of time performing verification. Every name character required several checks, and JDOM recognized a nonname character only after checking it first against every possible name character. Consequently,

Copyrighted material

the number of checks increased in direct proportion to the code point value. The project maintainers were beginning to grumble that maybe verification wasn't so important after all, and they might make it optional or ditch it entirely. Now, personally, I'm not willing to compromise correctness in the name of faster code, but it was apparent that the decision was going to be taken out of my hands if someone didn't do something. Fortunately, Jason Hunter did.

Hunter restructured my naïve code in a very clever way, shown in Example 5-5. Previously, even the common case where a character was legal required over 100 tests for each of the possible ranges of illegal characters. Hunter noticed that we could return a *true* value much sooner if we recognized both legal and illegal characters. This is especially beneficial in the common case where all names and content are ASCII, because these characters are the first ones we test.

EXAMPLE 5-5. Optimized digit character verification

```
public static boolean isXMLDigit(char c) {

    if (c < 0x0030) return false;  if (c <= 0x0039) return true;
    if (c < 0x0660) return false;  if (c <= 0x0669) return true;
    if (c < 0x06F0) return false;  if (c <= 0x06F9) return true;
    if (c < 0x0966) return false;  if (c <= 0x096F) return true;

    if (c < 0x09E6) return false;  if (c <= 0x09EF) return true;
    if (c < 0x0A66) return false;  if (c <= 0x0A6F) return true;
    if (c < 0x0AE6) return false;  if (c <= 0x0AEF) return true;

    if (c < 0x0B66) return false;  if (c <= 0x0B6F) return true;
    if (c < 0x0BE7) return false;  if (c <= 0x0BEF) return true;
    if (c < 0x0C66) return false;  if (c <= 0x0C6F) return true;

    if (c < 0x0CE6) return false;  if (c <= 0x0CEF) return true;
    if (c < 0x0D66) return false;  if (c <= 0x0D6F) return true;
    if (c < 0x0E50) return false;  if (c <= 0x0E59) return true;

    if (c < 0x0ED0) return false;  if (c <= 0x0ED9) return true;
    if (c < 0x0F20) return false;  if (c <= 0x0F29) return true;

    return false;

}
```

The earlier implementation checked a character against all possible digits, including such unlikely things as é and φ, before deciding a character wasn't a digit. The newer approach could determine more quickly that a character wasn't a valid digit. Similar and even more significant improvements were made to the checks for letters, extenders, and combining characters.

This didn't eliminate the time spent on verification, but it did reduce it enough that the project maintainers were appeased, at least for the case of element names. PCDATA verification still wasn't in the build, but that wasn't quite as big a problem.

Copyrighted material

Version 4: Second Optimization: Don't Check Twice

At this point, the time spent on verification had dropped by about a factor of four, and was no longer a huge concern. Version 3 is essentially what shipped in JDOM 1.0. However, by this point I had decided that JDOM was not good enough, and suspected that I could do better. My defection had more to do with issues of API design than with performance. I was also concerned with correctness, since JDOM still wasn't verifying everything it could, and it was still possible (though difficult) to use JDOM to create malformed documents. Consequently, I embarked on XOM.

XOM, unlike JDOM, made no compromises on correctness in the name of performance. The rule in XOM was that correctness came first, always. Nonetheless, for people to choose XOM over JDOM, its performance was going to have to be comparable to or better than that of JDOM. Thus, it was time to take another whack at the verification problem.

The optimization efforts of JDOM version 3 had improved the performance of the `checkXMLName` method, but I hoped to eliminate it completely in this next optimization. The reason for this is that you don't always need to check the XML input if it's coming from a known good source. In particular, an XML parser carries out many of the necessary checks before the input reaches the XML verifier, and there's no reason to duplicate this work. Because the constructors always checked for correctness, they caused a real drain on parsing speed performance, which in practice was a large fraction (often a substantial majority) of the time an application spent on each document.

The use of separate paths for separate types of input would resolve this issue. I had determined that constructors should *not* verify the element names when creating an object from strings that the parser had already read and checked in the document. Conversely, constructors *should* verify the element names when creating an object from strings passed by the library client. Clearly, two different constructors were needed; one for the parser and one for everybody else.

JDOM developers had considered this optimization, but got hung up on poor package design. In JDOM, the `SAXBuilder` class that creates a new `Document` object from a SAX parser is in the `org.jdom.input` package. The `Element`, `Document`, `Attribute`, and other node classes are in the `org.jdom` package. This means that all verifying and nonverifying constructors called by the builder must be public. Consequently, other clients can also call those constructors—clients that aren't making the appropriate checks. This enables JDOM to produce malformed XML. Later, in JDOM 1.0, the developers reversed themselves and decided to bundle a special factory class that accepted unverified input. This factory class is faster, but opens up a potentially troublesome backdoor in the verification system. The problem was just an artifact of separating the JDOM builder into input and core packages.

Copyrighted material

NOTE

Excessive package subdivision is a common anti-pattern in Java code. It often leaves developers faced with the unappealing choice of either making things public that shouldn't be, or limiting functionality.

Do not use packages merely to organize a class structure. Each package should be essentially independent of the internals of all other packages. If two classes in your program or library have to access each other more than they have to access other, nonrelated classes, they should be placed together in one package.

In C++, *friend* functions solve this problem neatly. Although Java does not currently have friend functions, Java 7 may make it possible to grant more access to subpackages that members of the general public do not have.

When I commenced work on XOM, I had the example of JDOM to learn from, so I kept the input classes in the same package as the core node classes. This meant I could provide package-protected, nonverifying methods that were available to the parser, but not to client classes from other packages.

The mechanics of XOM are straightforward. Each node class has a private no-args constructor, along with a package-protected factory method named `build` that invokes this constructor and sets up the fields without checking the names. Example 5-6 demonstrates this with the relevant code from the `Element` class. XOM is actually a little pickier than most parsers about namespaces, so it does have to check those. Still, it can omit a lot of redundant checks.

EXAMPLE 5-6. Parser-based digit character verification

```
private Element() {}

static Element build(String name, String uri, String localName) {

    Element result = new Element();
    String prefix = "";
    int colon = name.indexOf(':');
    if (colon >= 0) {
        prefix = name.substring(0, colon);
    }
    result.prefix = prefix;
    result.localName = localName;
    // We do need to verify the URI here because parsers are
    // allowing relative URIs which XOM forbids, for reasons
    // of canonical XML if nothing else. But we only have to verify
    // that it's an absolute base URI. I don't have to verify
    // no conflicts.
    if (! "".equals(uri)) Verifier.checkAbsoluteURIReference(uri);
    result.URI = uri;
    return result;

}
```


Copyrighted material

This approach dramatically and measurably sped up parsing performance, since it didn't require the same large amount of work as its predecessors.

Version 5: Third Optimization O(1)

After I implemented the constructor detailed in the previous section and added some additional optimizations, XOM was fast enough for anything I needed to do. Read performance was essentially limited only by parser speed and there were very few bottlenecks left in the document-building process.

However, other users with different use cases were encountering different problems. In particular, some users were writing custom builders that read non-XML formats into a XOM tree. They were not using an XML parser, and therefore were not able to take the shortcut that bypassed name verification. These users were still seeing verification as a hot spot, albeit a smaller one than it had been.

I wasn't willing to turn off verification completely, despite requests to do so. However, it was obvious that the verification process had to be sped up. The approach I took is an old optimization classic: *table lookup*. In a table lookup, you create a table that contains all the answers for all the known inputs. When given any input, the compiler can simply look up the answer in the table, without having to perform a calculation. This is an $O(1)$ operation, and its performance speed is close to the theoretical maximum. Of course, the devil is in the details.

The simplest way to implement table lookup in Java is with a `switch` statement. *javac* compiles this statement into a table of values stored in the byte code. Depending on the `switch` statement cases, the compiler creates one of two byte code instructions. If the cases are contiguous (e.g., 72–189 without skipping any values in between) the compiler uses a more efficient `tableswitch`. However, if any values are skipped, the compiler uses the more indirect and less efficient `lookupswitch` instruction instead.

> **NOTE**
> **This behavior isn't absolutely guaranteed, and may perhaps not even be true in more recent virtual machines (VMs), but it certainly was true in the generation of VMs I tested and inspected.**

For small tables (a few hundred cases or less), it was possible to fill in the intermediate values with the default value. For instance, a simple test can determine whether a character is a hexadecimal digit (Example 5-7). The test starts with the lowest possible true value, `'0'`, and finishes with the highest possible true value, `'f'`. Every character between `0` and `f` must be included as a case.

Copyrighted material

EXAMPLE 5-7. switch statement character verification

```
private static boolean isHexDigit(char c) {

    switch(c) {
        case '0': return true;
        case '1': return true;
        case '2': return true;
        case '3': return true;
        case '4': return true;
        case '5': return true;
        case '6': return true;
        case '7': return true;
        case '8': return true;
        case '9': return true;
        case ':': return false;
        case ';': return false;
        case '<': return false;
        case '=': return false;
        case '>': return false;
        case '?': return false;
        case '@': return true;
        case 'A': return true;
        case 'B': return true;
        case 'C': return true;
        case 'D': return true;
        case 'E': return true;
        case 'F': return true;
        case 'G': return false;
        case 'H': return false;
        case 'I': return false;
        case 'J': return false;
        case 'K': return false;
        case 'L': return false;
        case 'M': return false;
        case 'N': return false;
        case 'O': return false;
        case 'P': return false;
        case 'Q': return false;
        case 'R': return false;
        case 'S': return false;
        case 'T': return false;
        case 'U': return false;
        case 'V': return false;
        case 'W': return false;
        case 'X': return false;
        case 'Y': return false;
        case 'Z': return false;
        case '[': return false;
        case '\\': return false;
        case ']': return false;
        case '^': return false;
        case '_': return false;
        case '`': return false;
```


Copyrighted material

EXAMPLE 5-7. switch statement character verification (continued)

```
            case 'a': return true;
            case 'b': return true;
            case 'c': return true;
            case 'd': return true;
            case 'e': return true;
            case 'f': return true;
        }
        return false;
    }
```

This is long but shallow. It is not complex. It is easy to see what's happening here, and that's good. However, although `switch` statements are shallow, they do run into problems for larger groups of cases. For instance, XML character verification checks tens of thousands of cases. I tried writing a `switch` statement to handle these larger groups and discovered that Java imposes a 64K maximum size on the byte code of a method. This situation required an alternate solution.

Although the compiler and runtime limited the size of the lookup table stored in the byte code, there were other places I could hide it. I began by defining a simple binary format, one byte for each of the 65,536 Unicode code points in the Basic Multilingual Plane (BMP). Each byte contains eight *bit flags* that identify the most important character properties. For instance, bit 1 is *on* if the character is legal in PCDATA content, and *off* if it is not legal. Bit 2 is *on* if the character can be used in an XML name, and *off* if it cannot. Bit 3 is *on* if the character can be the start of an XML name, and *off* if it cannot.

I wrote a simple program to read the BNF grammar from the XML specification, calculate the flag values for each of the 65,536 BMP code points, and then store it in one big binary file. I saved this binary data file along with my source code, and modified my Ant compile task to copy it into the build directory (Example 5-8).

EXAMPLE 5-8. Saving and copying the binary lookup table

```
<target name="compile-core" depends="prepare, compile-jaxen"
          description="Compile the source code">
    <javac srcdir="${build.src}" destdir="${build.dest}">
       <classpath refid="compile.class.path"/>
    </javac>
    <copy file="${build.src}/nu/xom/characters.dat"
          tofile="${build.dest}/nu/xom/characters.dat"/>
</target>
```

From there, the *jar* task will bundle the lookup table with the compiled *.class* files, so it doesn't add an extra file to the distribution or cause any added dependencies. The `Verifier` class can then use the class loader to find this file at runtime, as shown in Example 5-9.

EXAMPLE 5-9. Loading the binary lookup table

```
    private static byte[] flags = null;

    static {
```


Copyrighted material

EXAMPLE 5-9. Loading the binary lookup table (continued)

```
        ClassLoader loader = Verifier.class.getClassLoader( );
        if (loader != null) loadFlags(loader);
        // If that didn't work, try a different ClassLoader
        if (flags == null) {
            loader = Thread.currentThread().getContextClassLoader( );
            loadFlags(loader);
        }

    }


    private static void loadFlags(ClassLoader loader) {

        DataInputStream in = null;
        try {
            InputStream raw = loader.getResourceAsStream("nu/xom/characters.dat");
            if (raw == null) {
                throw new RuntimeException("Broken XOM installation: "
                  + "could not load nu/xom/characters.dat");
            }
            in = new DataInputStream(raw);
            flags = new byte[65536];
            in.readFully(flags);
        }
        catch (IOException ex) {
            throw new RuntimeException("Broken XOM installation: "
              + "could not load nu/xom/characters.dat");
        }
        finally {
            try {
                if (in != null) in.close( );
            }
            catch (IOException ex) {
                // no big deal
            }
        }

    }
```

This task takes up about 64KB of heap space. However, that's not really a problem on a desktop or server, and we only have to load this data once. The code is careful not to reload the data once it's already been loaded.

Now that the lookup table is stored in memory, checking any property of any character is a simple matter of performing an array lookup followed by a couple of bitwise operations. Example 5-10 shows the new code for checking a noncolonized name, such as an element or attribute local name. All we have to do is look up the flags in the table and compare the bit corresponding to the desired property.

Copyrighted material

EXAMPLE 5-10. Using the lookup table to check a name

```
// constants for the bit flags in the characters lookup table
private final static byte XML_CHARACTER        = 1;
private final static byte NAME_CHARACTER       = 2;
private final static byte NAME_START_CHARACTER = 4;
private final static byte NCNAME_CHARACTER     = 8;

static void checkNCName(String name) {

    if (name == null) {
        throwIllegalNameException(name, "NCNames cannot be null");
    }

    int length = name.length( );
    if (length == 0) {
        throwIllegalNameException(name, "NCNames cannot be empty");
    }

    char first = name.charAt(0);
    if ((flags[first] & NAME_START_CHARACTER) == 0) {
        throwIllegalNameException(name, "NCNames cannot start " +
          "with the character " + Integer.toHexString(first));
    }

    for (int i = 1; i < length; i++) {
        char c = name.charAt(i);
        if ((flags[c] & NCNAME_CHARACTER) == 0) {
            if (c == ':') {
              throwIllegalNameException(name, "NCNames cannot contain colons");
            }
            else {
              throwIllegalNameException(name, "0x"
                + Integer.toHexString(c) + " is not a legal NCName character");
            }
        }
    }
}
```

Name character verification is now an $O(1)$ operation, and verification of a full name is $O(n)$, where n is the length of the name. You can fiddle with the code to improve the constant factors, as I have, but it's hard to see how this could be faster while still making the necessary checks. However, we're not done yet.

Version 6: Fourth Optimization: Caching

If you can't make the verification go any faster, the only remaining option is not to do it, or at least not do so much of it. This approach was suggested by Wolfgang Hoschek. He noticed that in an XML document the same names keep coming up over and over. For instance, in an XHTML document, there are only about 100 different element names, and a few dozen of those account for most elements (`p`, `table`, `div`, `span`, `strong`, and so on). Once you've verified that a name is legal, you can store it in a collection somewhere. The next time you see a name, you first check to see whether it's one of the names you've seen before; if it is, you just accept it and don't check it again.

Copyrighted material

However, you do have to be very careful here. It may take longer to find some names (especially shorter ones) in a collection such as a hash map than it would take to check them all over again. The only way to tell is to benchmark and profile the caching scheme very carefully on several different VMs using different kinds of documents. You may need to tune parameters such as the size of the collection to fit different kinds of documents, and what works well for one document type may not work well for another. Furthermore, if the cache is shared between threads, thread contention can become a serious problem.

Consequently, I have not yet implemented this scheme for element names. However, I have implemented it for namespace URIs (Uniform Resource Identifiers), which have even more expensive verification checks than do element names, and which are even more repetitive. For instance, many documents have only one namespace URI, and very few have more than four, so the potential gain here is much larger. Example 5-11 shows the inner class that XOM uses to cache namespace URIs after it has verified them.

EXAMPLE 5-11. A cache for verified namespace URIs

```
private final static class URICache {

    private final static int LOAD = 6;
    private String[] cache = new String[LOAD];
    private int position = 0;

    synchronized boolean contains(String s) {

        for (int i = 0; i < LOAD; i++) {
            // Here I'm assuming the namespace URIs are interned.
            // This is commonly but not always true. This won't
            // break if they haven't been. Using equals( ) instead
            // of == is faster when the namespace URIs haven't been
            // interned but slower if they have.
            if (s == cache[i]) {
                return true;
            }
        }
        return false;

    }

    synchronized void put(String s) {
        cache[position] = s;
        position++;
        if (position == LOAD) position = 0;
    }

}
```

There are a couple of surprising features in this class. First, rather than using the obvious hash map or table, it uses a fixed-size array with a linear search. For such small lists, the constant overhead of hash lookup is slower than simply iterating through the array.

Copyrighted material

INDEX

Symbols

$ (dollar sign)
  end of line matching in regular expressions, 3, 5
  in Perl variable names, 195
% (percent sign), Perl variable names, 195
& (ampersand), && operator in JavaScript, 136
( ) (parentheses)
  invoking functions, 144
  type in Haskell, 389
* (asterisk)
  JavaScript operator, 135
  matching zero or more occurrences in regular expressions, 3, 4
+ (plus sign)
  JavaScript infix operator, 134
  matching one or more instances in regular expressions, 43
. (period)
  . operator in JavaScript, 136
  matching any single character in regular expressions, 3
= (equals sign), === (exact-equality comparison) operator in JavaScript, 135
-> (arrow), indicating object-oriented method call in Perl, 195
? (question mark), ?: ternary operator in JavaScript, 135
@ (at-sign) in Perl variable names, 195
[ ] (square brackets)
  [ ] operator in JavaScript, 136
  [^ .], matching any character not a space or period, 43
  array literals in JavaScript, 144
^ (caret), beginning of line matching in regular expressions, 3
{ } (curly braces)
  delimiting code blocks in Ruby, 48
  enclosing function body in JavaScript, 143
  object literals in JavaScript, 145
  statement blocks in JavaScript, 141
| (vertical bar), || operator in JavaScript, 136
' (single quote) transposition operator, 233

Copyrighted material

Numbers

1-bits in a word, counting (see population count)
32-bit word, 147
  adapting HAKMEM population count for 36-bit word, 151
8086 assembly language (Windows 1.0), 111

A

A,G,C,T (nucleotides), 188
Acceptor wrapper facade, 437
accidental complexities encountered by developers, 429
ACE wrapper facades, 436
  socket wrapper facade, 442
ACE_Handle_Set member variables, 441
ACSS (Aural CSS), 511–516
  adding auditory icons, 512
  basic HTML using Emacs W3 and ACSS, 517
  data structure encapsulating ACSS settings, 512
  enhancing spoken output with context-sensitive semantics, 514–516
  mappings to TTS engines, 508
  producing auditory icons while speaking content, 514
ActionFixture class (Java), 77
actions (Haskell), 389
  important characteristics of, 404
  IO actions, 398
  main, 390
  STM actions, 392
  treated as first-class values, 391
  type system preventing IO actions running inside STM actions, 394
Adapter or Facade pattern, 8
additional levels of indirection, 279–291
  FreeBSD using indirection to abstract read function arguments, 282–285
  layering and, 290
  multiplexing and demultiplexing, 289
  operating system supporting different filesystems, 279
    filesystem layers, 285–287
    FreeBSD implementation of read system call, 280
advFilter method (Gene Sorter columns), 224
advice (Emacs LISP), 505
  auditory icons confirming user actions, 513
  definitions attached to calendar navigation functions, 515
  put-text-property function implementation, 508
  tutorial, 507
adviced function (LISP), 507
AFL (Audio Formatting Language), 511
after advice (Emacs LISP), 507
  auditory icons confirming user actions, 513
Ajax Generic Genome Browser, 211
alarm function (Perl), 171
ALGOL-like syntax for LISP, 130
algorithms
  customizing on the fly in Intermediate Language, 127
  data structures as frozen algorithms, 37
  for dense or sparse matrices, 230
  detailed in routine's description, 258
  search algorithms, 52
Amdahl's Law, 231
analysis of programs, 40
annotation, 188
  large number of genome annotation types, 190
anonymous subroutines (Perl), 207
Apache Commons HttpClient, 457
Apache Log4J Java package, 329
application level, science captured in mathematical models, 229
application-specific control structures, defining using Haskell actions, 404
area of a triangle, measuring, 548
around advice (Emacs LISP), 507
array descriptors, 246
array literals (JavaScript), 131, 144
arrays
  binary search
    boundary value testing, 92–94
    testing, 96
  bitmap pixels stored in, 115
  digital filter
    generating elements, 122
    storing byte in destination array, 126
    zero byte stored in destination array, 126
  digital image filters, 112
  JavaScript, selecting member with [ ] operator, 136
  N-dimensional, 303
  one-, two-, and three-dimensional, 303
  one-dimensional array and digital filter algorithm in C#, 116
  Perl variable names beginning with @, 195
  population count, 154–159
    CSA circuits, 154–159
    direct indexed access to moderately sparse array, 159
  in PyArrayIterObject, 312
  Python, free-dicts array of disused dictionary structures, 300
  Ruby implementation, 52
  (see also NumPy)

Copyrighted material

Art of Computer Programming, The, 37
assembly language, computation-intensive routines in, 266
assertions verifying the state of locks, 288
assignment operators (JavaScript), 137
association lists, 415
  assq procedure for lookups, 419
associations (Gene Sorter columns), 224
associative stores, 46–49
  using in dynamically typed languages, 47
AsTeR (Audio System For Technical Readings), 503
asynchronous event demultiplexer in reactive logging server, 442
Atom feeds, 522
atomically function (Haskell), 392
atomicity (Haskell transactions), 392
audio desktop (see Emacspeak audio desktop)
audio formatting codes, converting personality property to (Emacspeak), 510
Audio Formatting Language (AFL), 511
audio formatting using voice-lock (Emacspeak), 508
auditory icons, 512
  producing while speaking content, 514
  user interactions producing, 512
Aural CSS (see ACSS)
aural display lists
  audio-formatted output from, 510
  augmenting Emacs to create, 508–510
automata, regular expressions translated into, 45
automated debugging, 468
  further reading, 476
awk programming language, 50
  creations from FreeBSD read system call specification, 288

B

backtick function (Cryptonite), 181
backup tasks, MapReduce implementation, 380
Backus-Naur Form (BNF) grammar, checking XML names, 61, 63
balance in beautiful code, 481
bank accounts (concurrent programming example), 386–388
  using locks, 386
base categories (ERP5)
  predecessor, 349
  source_project and destination_project, 347
  source_project, retrieving tasks of a project, 348
base category objects (Portal Categories), 346
base pairs (bp), 188
  handling scale in genomic feature representation, 191
bases of DNA (A, C, G, or T), 218
Basic Linear Algebra Subprograms (see BLAS)
baton (Subversion), 17
beautiful code
  characteristics of, 266
  CIP (Collaborative Information Portal), 319
  CIP system architecture, 322–325
  code that works accurately and efficiently, 253
  lack of type checking in Linux kernel, 272
  NumPy iterators, 307
  object-oriented framework-based design, 443
  object-oriented networked software, 431
  parallel programs, simple, elegant, and mistake-free, 385
  scalability, 259
  test and example programs, 260
before advice (Emacs LISP), 507
Bell, Gordon, 31
Bentley, Jon, 29–40, 87
Berkeley Fast Filesystem (ffs), 283
binary MIME message composition, 179
binary search, 52, 87
  advantages and disadvantages, 54
  escaping the loop, 54
  "On the Goodness of Binary Search", 52
  postings, 55
  testing, 90–102
    boundary value tests, 92–95
    performance, 101
    random testing, 95–100
    smoke test, 91
binary search tree (Quicksort), cost of inserting an element, 38
binary-ness, testing, 101
binding interaction, DNA/protein, 206
binding powers (JavaScript tokens), 134
  + operator, 134
bindings, 415
  determining binding associated with an identifier, 418
  mapping to labels with environments, 418
  variable, capture of a variable reference by, 408–411
binomially distributed random integers, 160
Bio::Graphics module (Perl), 188–215
  design process, 192–209
    developer interaction with the module, 192–195
    dynamic options, 206–209
    object classes, choosing, 198–200
    option processing, 200–205
    options, setting, 196–198
    script that uses (example), 205–206
  design requirements, 190
  examples of output, 189
    typical image, 189

Copyrighted material

Bio::Graphics module (Perl) (*continued*)
extending, 210–214
adding new glyphs, 212
supporting publication-quality images, 211
supporting web developers, 210
Bio::Graphics::Glyph class, 198–200
factory( ) and option( ) methods, 204
option( ) method, changing for dynamic options, 208
subclasses, 204
Bio::Graphics::Glyph::Factory class, 201–205
dynamic option processing, 207
methods, 202
Bio::Graphics::Panel class, 192, 198
configuration options, 197
image_and_map( ) method, 210
png( ) method, 212
SVG images, 212
Bio::Graphics::Track class, 198
Bio::SeqFeature::CollectionI interface, 193
Bio::SeqFeature::Generic class, 206
Bio::SeqFeatureI interface, 214
bioinformatics, 187
genome annotation types, 190
SOAP-style web service architectures, 453
BioMoby project, 453
BioPerl
annotation databases, 193
Bio::Graphics module and, 188–192
Bio::SeqFeatureI interface, 214
(see also Bio::Graphics module)
bison, generating C code implementing high-level domain-specific language, 288
bit flags (binary lookup table), 70
bit shift operators (Java), 88
bit shifts for collisions in Python dictionaries, 299
BitBlt (bit block transfer) function (Windows 1.0), 106
8086 machine code instructions as subroutine, 111
C code implementing, 110
raster operations, 108
bitmaps
code dealing with, 110
digital filter applied to, 112
digital filter transforming source into destination bitmap, 113
ImageFilterTest program, 115
BLACS (Basic Linear Algebra Communication Subprograms), 243
Blandy, Jim, 16, 17
BLAS (Basic Linear Algebra Subprograms), 230, 237
Level-1 BLAS, 235
Parallel BLAS (PBLAS), 245–247
resources for further reading, 252
ScaLAPACK library, 243
blessed reference, 174
block size (NB), 239
block statement (JavaScript), 141
blocking
bank account transactions (example), 387
lack of modularity, 388
threads in Solaris, 354
blocking chain, 356
iterating over it coherently, 358
unblocking of thread blocked on synchronization primitive, 358
block-partitioned algorithms, 230
blocks of code (JavaScript), 141
blur filter, 113
BNF (Backus-Naur Form) grammar, checking XML names, 61, 63
Bonwick, Jeff, 355
boundary value tests, 91
binary search, 92–95
array, 92–94
target item location, 94
bound-identifier=? predicate, 424
bp (base pairs), 188
handling scale in genomic feature representation, 191
branch instructions (Intermediate Language), 120
Blt (branch if less than) and S (short branch), 123
labels, 121
Bray, Tim, 41–57
break statement (JavaScript), 142
breaking the code (beautiful tests), 95
brevity in code, 261, 478
elimination of redundancy, 479
leaving out unnecessary information, 478
broadcasting (in NumPy), 316, 317
BSD (see FreeBSD operating system)
BT (business template), 346
buffering, 290
_ _builtin_ _ module (Python), 294
business partners, integrating using REST, 451–462
exchanging data with e-business protocols, 457–462
assembling XML response, 462
parsing XML using XPath, 458–462
exposing services to external clients, 452–455
routing services using factory pattern, 456–457
business processes, representing in ERP5, 341
business template (BT), 346

Copyrighted material

C

c (character), matching in regular expressions, 3
C language
  code for applying a digital filter, 113
  code implementing filesystem layers, 288
  code packing function arguments into a single structure, 288
  compilation from specialized high-level domain-specific language to, 288
  do-while loop, 4
  early Windows applications, 111
  hexadecimals, 150
  N-dimensional arrays, looping over, 304
  object-oriented model of code in Linux driver model, 277
  object-oriented programming and, 8
  operating system (OS) APIs, 429
  Perl interface to libraries, 168
  pointers, 5, 271
    compact code in regular expression matcher, 9
  polymorphic object, creating, 221
  population count for 36-bit word, adapting to 32-bit word, 151
  population count, counting 1-bits in a word *x*, 148
  PyDict_SetItem( ) function, 300
  Python implementation (see CPython)
  special-purpose syntactic constructs, 407
  structures
    inheriting and manipulating in Linux kernel, 272
    PyDictObject, 295
    reference counting, 273–276
  syntactic abstraction mechanism, preprocessor macros, 408
C#, 112
  code to implement digital image filters, 112
  code written for best performance, examining with IL Disassembler, 115
  digital filter algorithm, 116
  image processing code, 114
  indirection in, 290
  Intermediate Language, generating and then executing, 118
C++
  cfront preprocessor, compiling C++ code into C, 288
  implementation of MapReduce word frequency counting example, 382
  inheritance and overloading, 261
  parameterized types, 432
C. elegans genome, 189
caching
  in eLocutor, 486, 495
    SaveReverse subroutine, 495
  namespace URIs after verification, 72
Calendar (Emacs), speech-enabling, 515
calendar-forward-week function, advice definition for, 515
callbacks in Bio::Graphics
  usefulness of, 209
  using for each option passed to add_track( ), 207
cancellations, handling in Subversion delta editor, 25
Cantrill, Bryan, 353–369
capture of a variable reference by a variable binding, 408–411
carry-save adder (CSA) circuits, 154–159
cart object (Gene Sorter), 220
cart variables (Gene Sorter)
  avoiding name conflicts, 221
  communication between column filtering methods, 224
CAs (Certification Authorities), 164
causalities, 347
CD-ROMs, ISO-9660 filesystem, 279, 281
CERN library, 254
  inner beauty of code, 261–266
    beauty in flow, 265
    brevity and simplicity, 261
    frugality, 262–265
  outer beauty of code, 255–260
Certification Authorities (CAs), 164
CGI scripts
  advantages and disadvantages, 219
  Gene Sorter, 219
  lifetime, 220
  long-term data storage, 220
  short lifetime, advantages of, 221
chain of blocked threads, 356
chaining, 298
  in vector machines, 231
character classes (in regular expressions), 7
children with disabilities, communication in classroom via software, 501
choose function (Haskell), 402
CIP (Collaborative Information Portal), 319–338
  Mars Exploration Rover (MER) mission and, 320
    mission needs, 321
  Middleware Monitor Utility, 335
  robustness, 336–337
    dynamic reconfiguration, 337
    hot swapping, 337

Copyrighted material

CIP (Collaborative Information Portal) (*continued*)
streamer service case study, 325–328
functionality, 325
reliability, 328–336
service architecture, 326
system architecture, code beauty of, 322–325
CISC computers, fundamental instructions, 147
class libraries, 433
client/server software
Emacspeak speech servers and Emacspeak client, 504
integrating RPG legacy systems with, 460
clipboard, 498
CMF (Content Management Framework), 340
implementation of structural part of applications with CMF Types, 342
overview, 342
CMF Types, 342
code and data, interplay between, 105
becoming mirror images of each other, 127
code complexity, managing over time, 523
code examples from this book, using, xx
code generation, on the fly, 105–127
FilterMethodCS, 116–118
FilterMethodIL, 119–127
graphics functions in Microsoft Windows 1.0, 106–112
code reuse, 228
ERP5, 351
promoted by good design and clear, concise code, 261
Collaborative Information Portal (see CIP)
collinearity problem (see computational geometry)
collision control
for overlapping genomic features, 191
for tracks in Bio::Graphics module, 195
collisions in Python dictionaries, 298
columnDb.ra files, containing column metadata, 222
column-oriented algorithms (LINPACK), 235
columns (Gene Sorter), 223
filtering, 224
command dispatcher in Cryptonite::Mail::Server, 170
comments
Haskell, 389
presenting full documentation for purpose of a routine, 258
commercial off-the-shelf (COTS) software, 322
commits, transaction log, 394
Common LISP macros, 408
common words (in eLocutor), 496
commonality/variability analysis, 434
logging server framework, 434
comparison count function (Quicksort), 33
comparison operators, Python data types providing versions of, 298
comparisons, counting for binary search, 101
compilation phase, regular expression processing, 7
compilers optimizing away indirection, 291
complexity of code, using simpler tools for complex problems, 481
computational geometry, 540–551
collinearity problem, 541–543
collinearity, testing by comparing slopes, 544–545
LISP, using, 540
river meandering model, 547
testing collinearity by triangle inequality, 545
computed solutions vs. mathematical equations, 253
computer architecture, effects on matrix algorithms, 230
concurrency, 385–405
bank accounts (example), 386–388
using locks, 386
different models, 432
locks, problems with, 387
sequential concurrency models in logging servers implementation, 439–444
STM (Software Transactional Memory), 388–405
implementing transactional memory, 394
Santa Claus problem (example program), 396–403
side effects and I/O in Haskell, 388–391
summary of basic STM operations, 396
transactions in Haskell, 392–393
condition variables
blocking in bank account transactions, 387
coordinating concurrent programs, 386
no support for modular programming, 388
connection/data event handling, Reactive_Logging_Server, 442
constants
hexadecimals in C, 150
JavaScript, 138
constructors (methods), checking for correctness in XML verifier, 66
container_of macro, 271
casting back from core kobject to main struct device, 274
Content Management Framework (see CMF)

Copyrighted material

content-addressable memory, 46–49
context-sensitive menu selection in eLocutor long click, 489–491
context-sensitive semantics, enhancing spoken output, 514–516
continuous testing, 467
conversions between s-expressions and syntax objects, 424
cookies
  CGI script data storage, 220
  key into user table for Gene Sorter, stored in persistent cookie, 220
core dump, analyzing for Solaris user-level priority inheritance bug, 360–366
COTS (commercial off-the-shelf) software, 322
CPython, 294, 301
  selecting storage function dynamically, 297
critical path, 248
Crockford, Douglas, 129–145
Crypt::GPG module (Perl), 173
  security audit, 180–182
Crypt::PGP5 module (Perl), 173
cryptographic software, 185
Cryptonite (mail system), 162–184
  basic design, 169
  code cleanup, Perl modules for jump to scalable product, 173
  decrypted messages, persistence of, 176
  design goals and decisions, 168
  functioning prototype, 172
  insights from development process, 162
  integrated interface for key management, 177
  Mail Daemon (cmaild), 171
  mail store, modifying to use IMAP as backend, 182
  revamping the mail store, 174
  securing the code, 178
  test suite, 172
  usability considerations, 166
Cryptonite::Mail::Config module (Perl), 179
Cryptonite::Mail::HTML package, 173
Cryptonite::Mail::Server, command dispatcher, 170
Cryptonite::Mail::Service class, 170
CSA (carry-save adder) circuits, 154–159
CSS (Cascading Style Sheets)
  CSS2, Aural CSS, 511
  early implementation by Emacs W3 browser, 517
  (see also ACSS)
Cunningham, Ward, 75
Cypherpunks, 185

D

\d, matching any digit in regular expressions, 43
data display debugger (ddd), 464
data management (MER Mission), 322
data movement through the memory hierarchy, 230, 239
data representation, 290
data structures as frozen algorithms, 37
data types
  C++ parameterized types, 432
  Haskell
    declaring, 398
    distinguishing actions from pure values, 404
  skipping declarations when unnecessary, 478
database code, prototyping in Perl, 173
database schemes, independence of Bio::Graphics from, 192
databases
  annotation, supported by BioPerl, 193
  ERP5 relational database, 341
  in operating system kernel for device naming, 268
  ZODB, 340
datum->syntax procedure (syntax-case), 412, 413
  converting s-expression to syntax object, 424
DBD::Recall module (Perl), 175
DBD::SQLite module (Perl), 173
DCWorkflow, 340
ddchange plug-in for Eclipse, 475
ddd (data display debugger), 464
de Carvalho, Rogerio Atem, 339–351
deadlock
  explicit locking code in bank account program, 387
  single-thread deadlock, 366
Dean, Jeffrey, 371–384
debugging, 463–476
  a debugger, 464
  delta debugging, 470–472
  finding failure cause automatically, 468
  hunting the defect, 473–475
    finding failure causes in program state, 473
    tracing failure-causing variables back to statements causing them, 475
  minimizing input, 472
    automated testing and splitting input into smaller pieces, 473
  prototype problem with fragile approach, 475

Copyrighted material

debugging (*continued*)
search problem, 467
determining if one change depends on another, 468
divide and conquer strategy, using, 468
systematic process of, 466
backward traversal of cause-and-effect chain for failure, 466
consistent, disciplined use of scientific method, 466
user-level priority inheritance bug in Solaris, 360–366
(see also testing)
declarations, skipping when unnecessary, 478
decompositional approach to solutions for dense linear systems, 232
decrypted email messages, persistence, 176
DECTalk Express, 504
define-syntax form, 411
defining new variables in a scope (JavaScript parser), 138
Delivery class, 347
DeliveryLine class, 343
delta debugging, 470–472
current implementations, 475
further reading, 476
minimizing program code, 473
on program states, 473
searching for failure causes in program input, 472
minimizing input, 472
delta editor (Subversion), 11–28
benefits of, 27
interface, 17–22
dense linear systems, decompositional approach to solutions, 232
dense matrix problem, 230
decompositional approach to solutions, 232
error analysis and operation count, 250
dentry structure (Linux), 276
depot, 356
design defects manifesting as bugs, 354
destination_project base category, 347
devfs filesystem, 268
device structure (Linux), 268
memory usage, 276
pointers to, 271
reference counting, 273
devices (on Linux), 267
no proper handling in persistent manner, 268
shutting down/powering up properly, 268
struct device as base class, 268
DGEMM routine, 239
DGETF2 routine, 239
bottleneck in computers with faster processors, 240
dictionaries, Python, 293–301
basic operations on, 293
C implementation of Python, 294
collisions, 298
iterations and dynamic changes, 300
keys and values of different data types in single dictionary, 294
keys not ordered, 294
passing keyword arguments to a function, 294
PyDictObject structure, 295
representing module contents, 294
resizing, 299
determining new table size, 299
special-case optimization for small hashes, 297
special-casing, when it's worthwhile, 297
diff tool, 465
digital filters, 112
algorithm in C#, 116
fast digital filter algorithm, 117
generating code for elements, 122
digital image filters
blur filter, 113
C code to apply digital filter (example), 113
sharpness filter, 113
direction (genomic features), 190
directory trees
differences, expressing, 16
version control and tree transformation, 12–15
disabled persons, software for
eLocutor, 483–500
Emacspeak audio desktop, 503–525
future directions, 500
dispatcher state of threads in Solaris, 358
distance procedure (Lisp), 546
distributed programming with MapReduce, 371–384
distributed grep, 375
distributed implementation (example), 377–380
execution overview, 377
distributed sort, 376
extensions to the model, 380
inverted index, 376
programming model, 374
reverse web-link graph, 376
term vector per host, 376
word count program (example), 371–374
divide and conquer strategy
debugging, finding cause of failure in gdb, 468
logical subdivision of tasks into subroutines, 265
population count, 149–151
recursive LU algorithm, 240

Copyrighted material

DNA sequences, 188
DNA/protein binding site, 206
do notation (Haskell), 389
  composing STM actions, 393
documentation, 227
  full documentation of routine's usage in the code, 258
document-centric paradigm (ERP5), 339
Don't Repeat Yourself (DRY principle), 479
Dongarra, Jack, 229–252
double-precision IEEE floating-point arithmetic, 546
do-while loop, 4
driver program for Map and Reduce functions (example), 374
driver routines, 261
DRY principle (Don't Repeat Yourself), 479
DTRSM routine, 239
Duff's Device, 11
duplication of information in code, eliminating, 479
Dybvig, R. Kent, 407–428
dynamic dispatch, 289
dynamic programming, 34
dynamic tree repopulation in eLocutor, 491
dynamically typed languages, using associative stores, 47
DynamicMethod class, 119
  Invoke method, 126

E

e-business protocols
  exchanging data with, 457–462
    assembling XML response, 462
    parsing XML using XPath, 458–462
  Rosettanet, 453
edge conditions, focusing on, 369
editing text in eLocutor, 497
editor operated with a single button, designing, 483–501
educating children with disabilities, communication software for, 501
egrep, 2
EJBs (Enterprise JavaBeans), 322
elegant code, evolution with hardware, 229–252
  effects of computer architectures on matrix algorithms, 230
  further reading, 252
  future directions for research, 251
  LAPACK DGETRF subroutine, 237–240
  LINPACK DGEFA subroutine, 235–237
  multithreading for multi-core systems, 247–250
  recursive LU, 240–243
  ScaLAPACK PDGETRF, 243–247
element names (XML 1.0), checking, 61
eLocutor, 483–500
  basic design model, 484–487
  efficiency of user interface, 500
  input interface, 487–499
    cache implementation, 495
    clipboard, 498
    common words and favorites, 496
    dynamic tree repopulation, 491
    long click, 489–491
    macros, 499
    retracing paths, 497
    searching, 499
    Templates and Replace, 494
    tree structure presenting options, 488
    typing buffer, editing, and scrolling, 497
    typing, simple, 493
  niche market for wider community, 484
  search function, 499
  web sites for download and discussion list, 500
Emacs
  Emacspeak, 500
  extension for one-button tree navigation, 501
  IDE for development of secure mail system, 168
  W3 web browser, 517, 525
Emacspeak audio desktop, 503–525
  Aural CSS (ACSS), using to style speech output, 511–516
  insights gained from implementing and using, 524
  managing code complexity over time, 523
  online information access, 516–522
    basic HTML with Emacs W3 and ACSS, 517
    feed readers, 522
    Web command line and URL templates, 520
    websearch module for task-oriented search, 517–520
  producing spoken output, 504
  speech-enabling Emacs, 505–516
    advice tutorial, 507
    first-cut implementation, 505
    generating rich auditory output, 507–511
    implementing event queue in speech server, 505
emacspeak-auditory-icon property, 514
emacspeak-calendar module, 515
emacspeak-calendar-entry-marked-p function, 515
emacspeak-calendar-speak-date function, 515
emacspeak-minibuffer-setup-hook function, 513

Copyrighted material

emacspeak-personality-voiceify-faces variable, 509
emacspeak-speak-line function, 505
emacspeak-url-template module, 520
emacspeak-url-template-fetch command, 521
emacspeak-w3 module, 517
emacspeak-w3-extract-table-by-match function, 519
emacspeak-websearch tool for accessing directions from Yahoo! Maps, 518
emacspeak-websearch-yahoo-map-directions-get-locations function, 518
email client, full-featured (Cryptonite), 167
encrypted mail client, persistence of decryption, 176
encrypted or signed messages, information about MIME structures, 175
encryption of email, 165
end position (genomic features), 190
Enterprise JavaBeans (EJBs), 322
enterprise resource planning systems, 339
  goals of, 340
  (see also ERP5)
enterprise system architecture, beauty of, 322
enterprise system for NASA Mars Rover (see CIP)
environments (syntax-case expansion algorithm), 418
  meta environment and runtime environment, exp procedure, 419
Equalizer, 483
ERP (enterprise resource planning), goals of, 340
ERP5, 339–351
  code in its raw state, web site, 351
  concepts that lay the basis for representing business processes, 341
  features to enhance programming productivity, 345
  object-to-relational mapping scheme, 341
  Portal Categories portal service, 345
  Project, 346–351
    coding, 347–351
  UBM (Unified Business Model), 341
  XML technologies, use of, 341
  Zope components used by, 340
  Zope platform, 342–346
    four-level structure representing system classes, 345
error analysis, 250
error bounds, 250
error-correcting codes, 159
errors
  communication mechanism in LAPACK SGBSV routine, 260
  recovery problems with locks, 388
essay, treating code as, 477–481
Euclid
  advice to student, 540
  algorithm for calculating greatest common divisor of two numbers, 540
  formula for area of a triangle, 549
European Organization for Nuclear Research (see CERN library)
evaluation stack (virtual) in IL, 120
event queue, implementing inside speech server, 505
example programs, importance of, 260
execution phase, regular expression processing, 7
exp procedure, 419
expansion algorithms
  hygienic macro expansion algorithm (KFFD), 410
  syntax-case, 413–425
    comparing identifiers, 423
    conversions, 424
    core transformers, 421–423
    creating wraps, 417
    expander, 419–421
    expander example, 425–427
    identifier resolution, 418
    manipulating environments, 418
    parsing and constructing syntax objects, 423
    producing expander output, 415
    representations, 414
    starting expansion, 424
    stripping syntax objects, 415
    structural predicates, 416
    syntax errors, 416
exp-core procedure, 421
Expect module (Perl), 180
exp-exprs procedure, 421
exp-if procedure, 421
exp-lambda procedure, 421
exp-let procedure, 422
exp-letrec-syntax procedure, 422
explicit formulation of hypothesis (in debugging), 466
exp-macro procedure, 420
exp-quote procedure (syntax-case), 421
expression function, 134
expression-based LISP macros, 408
expressions
  datum->syntax procedure used for arbitrary expressions in syntax-case, 413
  statements vs., 140
extend-wrap helper, 423
extensibility of software, 83
extension points, discovering, 513
external clients, exposing services to, 452–455

Copyrighted material

F

Factory class (Bio::Graphics::Glyph), 201–205
  dynamic option processing, 207
  make_glyph( ) method, 202
  option( ) method, 202
factory pattern, routing services with, 456–457
familiarity (of beautiful code), 479
FAT-32 filesystem for the USB stick, 279
fault tolerance, MapReduce implementation, 379
favorites in eLocutor (frequently used words), 496
Feathers, Michael, 75–84
features (genomic), 190
  density of, 191
  handling scale in visual representations, 191
feed reading software, 522
ffs (Berkeley Fast Filesystem), 283
fgrep, 2
file readers and file writers (CIP streamer service), 326
filesystems, operating systems supporting different, 279
  code to access filesystems, 280
  filesystem layers, 285–287
  FreeBSD use of indirection to abstract read function arguments, 282–285
Filter class, ApplyFilter( ) method, 116
filterControls method (Gene Sorter columns), 224
FilterMethodCS, 116
  optimizing, 117
FilterMethodIL, 119–127
  DynamicMethod instance, invoking, 126
filters, Gene Sorter, 224
finding the definition of a name (JavaScript parser), 139
finite automata, regular expressions translated into, 45
first-class values, Haskell actions as, 404
FIT (Framework for Integrated Test), 75–84
  challenge of framework design, 78
  classes, 76
    relationships among, 77
  documents serving as tests, 76
  HTML parsing, 80–83
  open framework, 79
  open style of development, benefits of, 83
Fixture class (Java), 77, 79
flex, generating C code implementing high-level domain-specific language, 288
flexibility of beautiful code, 481
floating-point arithmetic, IEEE double precision, 546
flow in beautiful code, 265
Fogel, Karl, 11–28
for loops
  looping over N-dimensional arrays, 304
  Python iterators as predicates, 305
forkIO function (Haskell), 390, 401
fork-join model of computation, 247
Fortran, 234
  BLAS (Basic Linear Algebra Subprograms), 252
  LINPACK package, 235
  required use of Fortran 90 with recursive LU, 240
forward web link graph, 376
frameworks
  applied to networked software, 430
  Framework for Integrated Test (see FIT)
  object-oriented, key concepts, 433
free_dicts array (Python), 300
FreeBSD operating system
  high-level I/O abstraction independence, 289
  implementation of read system call, 280
    filesystem-independent part, 281
  interface functions and data structures, language written in, 287
  read system call, functions to avoid code duplication, 290
  supporting different filesystems, abstracting read function arguments, 282–285
freedom from enforcement from tools (flexibility), 481
free-identifier=? predicate, 424
frequently used words in eLocutor, 496
frugality in beautiful code, 262–265
full-text searches, 55
fullword immediate, 152
function arguments abstracted to argument pointers, 282–285
function calls, keyword arguments in (Python dictionaries), 300
function pointers, used to dispatch a request to different functions, 289
functional decomposition of nontrivial software, problems created by, 430
functions
  JavaScript, 131, 143
  naming, understanding purpose from the name, 226
  reentrant, 227
  reusable, 228
fundamental instructions on RISC and CISC computers, 147

Copyrighted material

G

Gaussian elimination, 229–252
MATLAB tool, 233
GBrowse, 211
GD::SVG module (Perl), 212
gdb (GNU debugger), 464
debugging
running diff on source code, 465
search problem, 467
splitting changes into four or more subsets, 469
failure caused by changes in new release, 464
searching program states for failure cause, 473
gene expression columns (Gene Sorter)
filtering, 224
Gene Sorter, 217–228
beautiful code, 225–228
function filtering associations that handles wildcards, 226
dialog with user over the Web, 219
filtering down to relevant genes, 224
genome.ucsc.edu implementation, 221
polymorphism, 221–224
user interface, 218
Generic class (Bio::SeqFeature), 206
genes, 218
genome browsers, web-based, 211
genome map rendering module (see Bio::Graphics module)
genome, annotation of, 188
genome.ucsc.edu implementation of Gene Sorter, 221
gen-var helper, 415
GET and POST requests over HTTP, services exposed through, 452
GFS (Google File System), 377
management of input data, 379
paper about design and implementation, 381
GHC (Glasgow Haskell Compiler), 403
Ghemawat, Sanjay, 371–384
Glasgow Haskell Compiler (GHC), 403
Glyph class (Bio::Graphics)
dynamic option processing, 208
factory( ) and option( ) methods, 204
subclasses, 204
glyphs (Bio::Graphics module), 196
adding new, 212
box model, 215
dynamic creation of, 200–205
Glyph class, 198–200
subglyph generation, 214
GNU debugger (see gdb)
GnuPG, 173
Google
high-quality search results, 56
MapReduce programs, number of, 376
MapReduce, development for large-scale computations, 371
(see also MapReduce)
Google Maps, Emacspeak tool with similar functionality, 519
Google News searches via Atom feeds, Emacspeak URL template for, 522
goto instruction, 120
GPG, security audit of Crypt::GPG module, 180–182
graphic formats, independence of Bio::Graphics from, 191
graphics function in Windows 1.0 (see BitBlt function)
grep, 2
distributed, 375
groupings of words (eLocutor), 486
Gulhati, Ashish, 161–186

H

HAKMEM memo, 151
Hamming distance between two bit vectors, 159
hardcoded parameter values, avoiding, 337
hardware interfaces, 290
hardware speech synthesizer (DECTalk Express), 504
hardware, evolution of elegant code with, 229–252
effects of computer architecture on matrix algorithms, 230
LAPACK DGEFR subroutine, 237–240
LINPACK DGEFA subroutine, 235–237
further reading, 252
future directions for research, 251
multithreading for multi-core systems, 247–250
recursive LU, 240–243
ScaLAPACK PDGETRF, 243–247
Harold, Elliotte Rusty, 59–74
hash tables, 46
binary search vs., 54
bitmask representing size in PyDictObject structure, 296
keys hashing to same slot, 298
loading a big hash, 50
resizing for Python dictionaries, 299
hashes
constructed by code reading columnDb, 223
filtered genes in Gene Sorter, 225
Ruby, 47
unordered hash in Perl, 195

Copyrighted material

Haskell, 386, 388–405
Glasgow Haskell Compiler (GHC), 403
side effects and I/O
functional nature of Haskell, 391
side effects and input/output, 388–391
actions, 389
summary of basic STM Haskell operations, 396
Hawking, Stephen, 483
Hayes, Brian, 539–551
hidden CGI variables, 220
key into session table for Gene Sorter, 220
Hoare, C. A. R., 30, 31, 37
hook methods, 435
in Logging_Server run( ) template method, 439, 443
use in reactive logging server implementation, 442
host infrastructure middleware, 430
hot swapping, 337
HotkeyAdaptor interface (example), 456
HotkeyAdaptorFactory class (example), 456
hourglass glyph (example), 213
HTML
FIT (Framework for Integrated Test), 76
rendered by speech-enabled Emacs W3 and ACSS, 517
HTTP exchanges, managing with Apache Commons HttpClient, 457
HTTP POST method, sending/receiving XML documents, 453
human genome sequence, 188
human memory, limiting factor in programming, 225
hygiene condition for macro expansion, 409
hygienic macro expansion
KFFD algorithm, 410
lexical scoping implemented for source code, 410
solving variable capture problems, 409

I

I/O abstraction independence on FreeBSD, 289
I/O in Haskell, 389–391
forkIO function, 390
IBM Java function library to integrate RPG systems with client/server software, 460
IBM WebSphere application server, Java Servlet running on, 453
icons, auditory (see auditory icons)
ideal partitioning method (Quicksort), 38
identifier? predicate, 416
identifiers
comparing, 423
determining binding associated with, 418
mapping to bindings with substitutions, 414
Python, built-in, 294
unintended variable captures
introduced bindings and references, 409
scoping in output instead of input, 408
id-label procedure, 418
if form, 421
if statement (JavaScript), 142
IL (see Intermediate Language)
IL Disassembler, 115
ILGenerator class, 119
Emit method, 120
image filters, 112
image maps for Bio::Graphics output, 210
image processing code, generating on the fly, 105–127
FilterMethodCS, 116–118
FilterMethodIL, 119–127
graphics functions in Microsoft Windows 1.0, 106–112
ImageFilter class, 115
images held in NumPy array, cropping and shrinking, 304
IMAP
modifying Cryptonite mail store to use, 182
performance bottlenecks in Cryptonite mail store after implementing, 184
IMAP server as a mail store (Cryptonite), 175
in situ debugging, 360
inaudible (Emacspeak personality), 511
include form, defining in syntax-case, 413
indexing feature, VB TreeView control, 492
indexing, matrix, 233
indirection, levels of (see additional levels of indirection)
individual rights in the digital age, 161
protection with communications privacy, 185
infix function (JavaScript), 135
infix operators (JavaScript), 134–136
inheritance structure, complicating code, 261
inheritance, JavaScript objects, 131
inode structure (Linux), 276
<INPUT> tags (HTML) of type hidden, 220
input interface (eLocutor), 487–499
cache implementation, 495
clipboard, 498
common words and favorites, 496
dynamic tree repopulation, 491

Copyrighted material

input interface (eLocutor) (*continued*)
long click, 489–491
context-sensitive menu selection, implementing, 489–491
macros, 499
retracing paths, 497
searching, 499
simple typing, 493
Templates and Replace, 494
tree structure presenting options, 488
typing buffer, editing, and scrolling, 497
input validation (Cryptonite mail system), 179
instance method calls in Java, 290
instructions (fundamental), RISC and CISC computers, 147
integrating business partners (see business partners, integrating using REST)
integration as key to success in large applications, 324
intended variable captures, 411
interactive web applications, use of Bio::Graphics output, 191, 210
Intermediate Language (IL), 112
algorithm customization on the fly, 127
assignment and operational logic based on virtual evaluation stack, 120
dynamic generation of, 119
function optimization for fast digital filter, 117
generated by C# compiler, 115
jump or branch instruction (goto), 120
interrupts, reasons for, 159
intersections of segments, 547
intonation in speech, 506
inventor's paradox, 33
inversion of control in runtime architecture of OO framework, 434
inverted index, constructing with MapReduce, 376
IO actions (Haskell), 398
list comprehension, used to create list of IO actions, 400
situations where use is essential, 400
STM (IO ( )) action type, 403
STM actions vs. as function types, 400
IPC (interprocess communication) mechanisms, 431
ACE wrapper facades, 436
associating concurrency strategy with, 437
IPC::Run module (Perl), 181
ISO-9660 filesystem, initializing vop_vector structure, 280
isolation (Haskell transactions), 392
isXMLDigit( ) method, 64
items (ERP5), 341
iterating over entire blocking chain coherently (in Solaris), 357
iteration, use by NumPy for N-dimensional algorithms, 305
iterations through Python dictionaries, 300
Iterative_Logging_Server, 440
evaluating, 443
Iterator pattern, instance implemented for socket handle sets, 442
iterators
designing in NumPy, 307–313
iterator counter tracking, 310
iterator progression, 308
iterator setup, 309
iterator structure, 312
iterator termination, 309
iterator interface in NumPy, 313
NumPy, origins of, 307
use in NumPy, 314–318
anecdotes, 317
iteration over all but one dimension, 315
multiple iterations, 316–317

J

J2EE (Java 2 Enterprise Edition), 322
reliability, 325
scalability, 325
SOA based on, in CIP, 323
Java
Apache Log4J package, 329
API to access functions running on AS/400 server, 453
arrays, 52
binary search algorithm bug, 88
binary search implementation, 52
binary search program (example), 52
class libraries for network programming, 430
client and data tier of CIP, 322
conversion of regular expression matcher to, 8
Framework for Integrated Test (see FIT)
Hello World program, 478
indirection in, 290
library for integrating RPG legacy systems with client/server software, 460
Python implementation (Jython), 297
random-number generator and Arrays utilities, 96
Session class, 78
Store class, 79
table lookup, implementation with switch statement, 68
(see also CIP)

Copyrighted material

Java Servlet running on IBM WebSphere application server, 453
JavaScript, 130–145
  array and object literals, 144
  assignment operators, 137
  constants, 138
  expressions, 134
  functions, 143
  infix operators, 134–136
  precedence, 133
  prefix operators, 136
  scope, 138–140
  Simplified, 130
  statements, 140–143
  symbol table, 131
  tokens, 132
  web sites creating smart client-side interaction via, 520
JDOM, 60
  bad package designs, 66
  name character verification, first version, 62
JSLint, parsing technique (example), 145
jump or branch instruction (Intermediate Language), 120
JUnit, 89
Jython, 297

K

Kent, Jim, 217–228
kernel synchronization primitives, priority inheritance (Solaris), 356
Kernighan, Brian, 1–9
key authentication
  PKI vs. web of trust, 186
  public-key cryptography, 164
key management, integrated interface in Cryptonite, 177
key/expression pair (in JavaScript object literals), 145
key/value pairs, 46
  Python dictionaries, 293
keys
  Python dictionaries
    collisions, 298
    different data types in single dictionary, 294
    looking up, 297
    not ordered, 294
keyword arguments in function calls (Python dictionaries), 300
keyword bindings, 415
keywords
  keyword binding, associating the keyword or with a transformer, 411
  reserving the names of, 409
KFFD (hygienic macro expansion algorithm), 410
Kleene, Stephen, 1
Knuth, D. E., 37
kobject structure (Linux), 274
  changed to use struct kref, 275
Kolawa, Adam, 253–266
kref structure (Linux), 275
Kroah-Hartman, Greg, 267–277
Kuchling, Andrew, 293–301

L

labels, 415
  determined by marks and substitutions in an identifier's wrap, 418
  in Intermediate Language, 121
  mapping to bindings with environments, 418
lambda expressions
  in define-syntax form, 411
  exp-lambda procedure, 421
Lampson, Butler, 279
language independence in CIP, 324
LAPACK, 237–240
  LU factorization, 238
  resources for further reading, 252
  SGBSV routine, 255–260
    implementation details, 261
  SGBTRF routine, 262–265
  (see also ScaLAPACK)
large applications, beauty in, 338
large-scale data processing problems, programming system for (see MapReduce)
layering and indirection, 290
layering of filesystems, 285–287
  on FreeBSD, 282
led (left denotation) method, 134
let form, 422
letrec-syntax forms
  exp-letrec-syntax transformer, 422
letrec-syntax, binding macros within a single expression, 412
let-syntax form, 422
Level-1 BLAS, 235
  use in LINPACK factorization, 236
Level-2 BLAS, 230, 243
Level-3 BLAS, 230, 243
levels of indirection (see additional levels of indirection)
lex (lexical analyzer generator), mapping regular expressions into actions, 288
lexical scoping, implementation by hygienic macro expansion, 410
lexical variable bindings, 415
libgd, 211

Copyrighted material

likeliest causes first, examining in debugging, 467
linear algebra
algorithms recast as matrix-matrix operations, 231
core of scientific computing calculations, 229
dense linear systems, decompositional approach to solutions, 232
software for advanced-architecture computers, 229
motivation for development, 230
linear probing, 298
line-segment intersection algorithms, 547
link counting (PageRank), 56
LINPACK, 235–237
column-oriented algorithms, use of, 235
implementation of factorization, 235
resources for further reading, 252
Linux
desktop environment for secure mail system, 168
native filesystem, 279
Linux kernel, 267–277
development process, how it works, 277
devfs, problems with race conditions, 268
device handling in persistent manner, lack of, 268
devices, physical and virtual portions, 267
power management for devices, 268
unified driver and device model, 268
object reference counting in virtual filesystem layer, 273–276
pointers to struct device, passing around, 271
runtime type checking, lack of, 272
scaling up to thousands of devices, 276
small objects loosely joined, 277
struct device as base class for all devices, 268
sysfs virtual filesystem, 269
LISP
advice, 505
extension points, discovery of, 513
tutorial, 507
ALGOL-like syntax, attempts at, 130
macros, 408
parsing techniques, 130
procedure definition, 540
list comprehension (Haskell), 400
literals (in JavaScript), 131, 144
literal symbol in JavaScript parser, 132
load balancing, MapReduce implementation, 379
loadable parameters for services, 337
locality, MapReduce implementation, 379
locking, 287
bank accounts using locks, 386
handling of locking assertions, 288
Mutex wrapper facade for acquiring/releasing locks, 437
problems with locks, 387
error recovery, 388
lost wakeups and erroneous retries, 388
no support for modular programming, 388
taking in wrong order, 388
taking the wrong locks, 388
taking too few, 387
taking too many, 387
Solaris user-level priority inheritance bug, 354–368
strategies for, 432
Log_Handler class, 438
logging
CIP streamer service, 329–334
concurrent logging servers, implementing, 444–450
log record formats, 432
networked logging service (example application), 431
Sawzall language for logs analysis, 382
sequential logging servers, implementing, 439–444
logging server framework, OO design for, 432, 433–439
associating concurrency strategy with IPC and synchronization, 437
commonalities, understanding, 434
key concepts about OO frameworks, 433
variation, accommodating, 435
Logging_Server abstract base class, 432, 437
open( ) and request( ) methods, 438
run( ) method, 439
logical index into an array, translating to physical index, 160
logical operators (short-circuiting), in JavaScript, 136
longest match, 5
leftmost longest matching, matchstar function, 6
look-ahead computations, 248
lookdict( ) and lookdict_string search functions, 300
lookup table
binary format, 70
loading, 70
using to check a name, 71
Java instance method calls dispatched through, 290
lookup type (Gene Sorter columns), 223
loops, escaping in binary search, 54

Copyrighted material

loose coupling of client applications and middleware services, 324
LU factorization, 230
  error analysis and operation count, 250
  LAPACK SGBTRF routine, 262–265
  LAPACK solution, 238
  multithreaded, 247–250
  recursive, 240–243
  ScaLAPACK solution, 243–245
  simple implementation, 233
Luszczek, Piotr, 229–252

M

m4 macro expander, 408
Mac OS X, development platform for secure mail system, 168
macros
  eLocutor, 499
  offsetof macro, 271
  for syntactic abstraction, types of, 408
  syntax-case macro definition, 411
  (see also syntactic abstraction)
magazines, 356
Mail Daemon (cmaild), Cryptonite, 171
  test suite, 172
mail system (Cryptonite), 162–184
  insights from development process, 162
  mail store, 173
    modifying to use IMAP as backend, 182
    performance bottlenecks with IMAP, 184
    replication, 175, 183
    revamping, 174
Mail::Cclient module (Perl), 184
Mail::Folder module (Perl), 175
Mail::Folder::Shadow module (Perl), 176
Mail::IMAPClient module (Perl), 184
main (I/O action in Haskell), 390
main action (Haskell), 401
Mak, Ronald, 319–338
managed code, 112
mapping (Python), 296
MapReduce, 371–384
  computations easily expressed as MapReduce, 375
  distributed implementation (example), 377–380
    execution overview, 377
  extensions to the model, 380
  programming model, 374
  resources for further reading, 381
  word count program (example)
    C++ implementation, 382
mark-object auditory icon, 514
marks, 413
  appearing in an identifier's wrap, determining associated label, 418
Mars Exploration Rover mission (MER), enterprise system for (see CIP)
match function, 4
matchhere function, 4
  longest matching implementation, 6
matchstar function, 5
  leftmost longest matching, 6
mathematical equations vs. computed solutions, 253
mathematical models, science at the application level, 229
MATLAB, 233
matrix algorithms, 230
  computer architecture effects on, 230
  development with MATLAB, 233
  expressing as vector-vector operations, 231
matrix computations through decomposition, 232
matrix functions built into MATLAB, 233
matrix-matrix operations
  Level-3 BLAS, 230
  linear algebra algorithms recast as, 231
  modularity for performance and transportability, 239
matrix-vector operations
  Level-2 BLAS, 230
  recasting linear algebra algorithms as, 237
  recasting linear algebra in terms of, 231
Matsumoto, Yukihiro, 477–481
mbox files, Cryptonite, 173
Mehta, Arun, 483–501
memory
  check looking at problem size and memory of computer, 265
  conservation of, 258
  content-addressable, 46–49
  usage by binary search, 54
  use of (LAPACK SGBSV routine), 260
memory models for N-dimensional array, 305
menu selection (context-sensitive), in eLocutor long click, 489–491
messenger RNA (mRNA), 218
Meta Classes (ERP5), 343
meta environment (exp procedure), 419
metacharacters (regular expressions), 1, 7
metadata in filesystem layering, 287
method form (syntax-case), 413
Microsoft Intermediate Language (MSIL) (see Intermediate Language)
Microsoft Windows (version 1.0), on-the-fly code generation, 106–112

Copyrighted material

middleware
host infrastructure, 430
in multitiered service-oriented architecture, 320
services loosely coupled with client applications, 324
stateless session beans as service providers, 323
Middleware Monitor Utility (CIP), 335
middleware.properties file, 337
midpoint calculation, binary search, 92
calculateMidpoint( ) method, boundary test for, 94
MIME structure of Cryptonite mail messages, 175, 183
modular programming
concurrent programming using STM, 404
not supported by locks and condition variables, 388
parallel programs less modular, 385
modularity in CIP, 324
modules (Python), contents represented as a dictionary, 294
modulus function, unsigned, 152
Monnerat, Rafael, 339–351
monotone voice with no inflection (Emacspeak), 512
motor disabilities, software designed to accommodate, 483–501
Movement class (ERP5), 342
representing cash withdrawal and material transfer, 345
movements (ERP5), 341
mRNA (messenger RNA), 218
MSIL (Microsoft Intermediate Language) (see Intermediate Language)
multi-core systems, 231
effects on matrix algorithms, 231
multithreading for, 247–250
multidimensional iterators (see NumPy)
multithreaded programs in C, reference counting for structures, 273–276
multithreaded, multi-core processors, data consistency through locking, 287
multitiered service-oriented architecture, 320
mutable variables, reading/writing (Haskell side effect), 390
Mutex wrapper facade, 437

N

naïve-collinear function, 542
names
single style convention, importance of, 227
validation in XML, 60
well-chosen, importance in making code understandable, 226
namespace URIs, caching after verification, 73
NASA's Mars Rover mission, enterprise system for (see CIP)
navigation in eLocutor, 497
NB (block size), 239
N-dimensional arrays, 303
key challenges in operations, 304
memory models for, 305
(see also NumPy)
Neomailbox secure email service, 177
.NET Common Language Runtime, 112
.NET Intermediate Language, 117
networked software, object-oriented framework for, 429–450
design of logging server framework, 433–439
commonalities, 434
key concepts about OO frameworks, 433
tying it all together, 437
variations, 435
implementing concurrent logging servers, 444–450
implementing sequential logging servers, 439–444
logging service application, 431
next word (in eLocutor), 493
NFS (Network File System), interposing umapfs over, 285
nodes (ERP5), 341
noncontiguous arrays, 306
notation, regular expression, 2
NPR programs, URL template for (Emacspeak), 521
nucleotides (A,G,C,T), 188
nud (null denotation) method, 134
null match, 4
Null Mutex, 443
Null Object pattern, 437
Null_Mutex facade, 440
nullfs filesystem, implementing the bypass function, 284
NumPy (Python), 303–318
accessing any region of an array using slicing, 304
iterator design, 307–313
counter tracking, 310
progression, 308
setup, 309
structure, 312
termination, 309
iterator interface, 313
iterator origins, 307
iterator use, 314–318
anecdotes, 317
iteration over all but one dimension, 315
multiple iterations, 316–317

Copyrighted material

memory models for N-dimensional array, 305
N-dimensional array operations, key challenges in, 304
use of iteration for N-dimensional algorithms, 305

O

object classes (Bio::Graphics), 192
choosing, 198–200
object literals (JavaScript), 131, 145
object reference counting in Linux kernel virtual filesystem layer, 273–276
object-caching allocator, 356
object-oriented (OO) programming languages
dynamic dispatch to various subclass methods, 289
frameworks for networked software, 430
object-oriented framework for networked software, 429–450
design of logging server framework, 433–439
commonalities, 434
key concepts about OO frameworks, 433
tying it all together, 437
variations, 435
implementing concurrent logging servers, 444–450
implementing sequential logging servers, 439–444
logging service application, 431
object-oriented programming
C language and, 8
method call in Perl, 195
objects
dynamic, with prototypal inheritance (JavaScript), 131
polymorphic, in Gene Sorter, 221–224
Objects/dictnotes.txt file, 301
Objects/dictobject.c source file, 301
Observer pattern, 434
offsetof macro, 271
Oliphant, Travis E., 303–318
one-dimensional arrays, 303
online information access with Emacspeak (see web-oriented tools in Emacspeak)
OpCodes class, 120
open addressing, 298
OpenBSD, development platform for secure mail system, 168
OpenPGP, 164
communicating with in Cryptonite mail system, 181
key operations made available to users, 166
MIME-aware mail store, 175
security embedded in Cryptonite, 165
operating systems
C language APIs, 429
database or devfs to handle device naming, 268
locks for critical operating system structures, 287
supporting different filesystems, 279
code to access filesystems, 280
filesystem layers, 285–287
FreeBSD using indirection to abstract read function arguments, 282–285
synchronous event demultiplexing APIs, using with reactive logging server, 441
operation count, 251
operator precedence, 129–145
JavaScript, 130–145
array and object literals, 144
assignment operators, 137
constants, 138
expressions, 134
functions, 143
infix operators, 134–136
precedence, 133
prefix operators, 136
scope, 138–140
statements, 140–143
symbol table, 131
tokens, 132
top down, 129
optimistic execution, 394
optimization
base-case optimization for recursive code, 305
XML verifier
caching namespace URIs after verification, 72
digit character verification, 65
lookup table, 68–72
parser-based digit character verification, 66
Order class (ERP5), 347
source base category, 346
order_validateData script, 349
ordering guarantees (MapReduce), 380
OS X, development platform for secure mail system, 168
Otte, William R., 429–450
overloading, leading to complicated code, 261
ownership of locks, tracking for user-level locks, 363

Copyrighted material

P

package designs, Java, 66
PageRank, 56
pairs
  key/expression, in JavaScript object literals, 145
Panel class (Bio::Graphics), 192, 198
  configuration options, 197
  image_and_map( ) method, 210
  png( ) method, 212
  SVG images, 212
Parallel BLAS (PBLAS), 245–247
  array descriptors, 246
parallel programs, 385
parallel systems with distributed memory, 231
  effects on matrix algorithms, 231
parallelism
  application in web searches, 56
  (see also concurrency)
parallelized word count program (example), 372
  with partitioned processors, 373
  with partitioned storage, 372
Params::Validate module (Perl), 179
Parse class (Java), 77, 80–83
  doCells( ) method, 80
  doRows( ) method, 80
  doTables( ) method, 79
  ignore method, 80
  last( ) and more( ) methods, 83
  parsing code, 81
  representation of entire HTML document, 82
parser-based digit character verification for XML, 66
parsers
  JavaScript, 130–145
    array and object literals, 144
    assignment operators, 137
    constants, 138
    expressions, 134
    functions, 143
    infix operators, 134–136
    precedence, 133
    prefix operators, 136
    scope, 138–140
    statements, 140–143
    tokens, 132
  top-down operator, 129
  XML, checking for correctness in XML input, 66
parsing techniques in LISP, 130
partitioned processors for parallelized program, 373
partitioned storage for parallelized program, 372
partitioning an index based on binary search in arrays of postings, 56
partitioning function (MapReduce), 380
partitioning in Quicksort, ideal method, 38
paths
  ERP5, 341
  retracing in eLocutor tree structure, 497
Pattern and Matcher classes (Java), 8
pattern matching
  in regular expressions (see regular expressions)
  syntax form in syntax-case, 411
pattern or brush (graphical object), 108
patterns
  applied to networked software, 430
  event-dispatching, 434
  frequently encountered in computer programming, 407
  used in OO logging server framework, 432
Patzer, Andrew, 451–462
PBLAS (Parallel BLAS), 245–247
  array descriptors, 246
PCI and USB devices in Linux sysfs, 269
performance
  indirection and, 291
  poor design as root cause of problems, 259
  Python dictionary implementation and, 294
  recursion and, 305
  regular expression matcher, 6
  testing for binary search, 101
  XML verifiers, correct design vs., 74
Perl, 50
  alarm function, 171
  anonymous subroutines, 207
  AUTOLOAD feature, 176
  BioPerl, Bio::Graphics module and, 188–192
  Crypt::PGP5 module, 173
  Cryptonite mail system, 168
  Emacs's cperl mode, 168
  GD library, 211
  GD::SVG module, 212
  prototype-to-production path, DBD::SQLite module, 173
  summary of quirkier parts of syntax, 195
  Text::Template module, 173
  Tie interface, used to tie Postgres' large objects (BLOBs) to filehandles, 174
Persistence::Database::SQL class, 173
Persistence::Object::Postgres class, 173
Persistence::Object::Simple class, 173
personality text properties (Emacspeak), 508
  converting into audio formatting codes, 510
personnel management (MER mission), 322

Copyrighted material

Petzold, Charles, 105–127
Peyton Jones, Simon, 385–406
PGP, 164
encryption backend for Cryptonite mail, 173
physical index into an array, translating logical index to, 160
pitch setting for flat voice (Emacspeak), 512
pitch-range setting (Emacspeak), 512
pivots, 232
PKI (Public Key Infrastructure)
reimposition of authoritarian cultural tendencies, 186
shortcomings of, 164
pointers, 5
contribution to compact code in regular expression matcher, 9
function arguments to argument pointers, 282–285
function pointers dispatching a request to different functions, 289
function pointers, isolating filesystem implementation from code accessing its contents, 282
PyDictEntry structures, 296
runtime type checking lacking on Linux kernel, 272
to a struct device (on Linux), 271
to vop_vector structure, 280
polymorphism in Gene Sorter, 221–224
pop method, closing a scope in JavaScript parser, 139
population count, 147–160
basic methods, 148
comparing for two words, 153
counting 1-bits in an array, 154–159
divide and conquer strategy, 149–151
HAKMEM memo, 151
sum and difference of two words, 152
uses of, 158
uses of population count instruction, 158
portal (CMF), 340
Portal classes, 344
portal_types service, 342
POST and GET requests over HTTP, services exposed through, 452
Postgres database, object persistence, 173
postings, searching with, 55
postmortem debugging, 360
Practice of Programming, The, 2
Pratt, Vaughan, 129
pre-action and post-action hooks, 517
precedence, 133
(see also operator precedence)
precedence levels (see binding powers)
predecessor base category, 349
prediction
in eLocutor dynamic tree repopulation, 491
evaluating efficiency of for typing in eLocutor, 500
word completion and next word in eLocutor, 493
prefix operators (in JavaScript), 136
preprocessor macros (C), 408
priority inheritance, 356–368
getting it correct, 357
implementing for user-level locks, 363
user-level locks
waiving, 366
user-level, debugging, 360–366
priority inversion, 355
privacy in communications, protecting individual rights, 185
private keys, user identify tied to in Cryptonite, 167
processor-independent Intermediate Language (see Intermediate Language)
product-line architecture, 432
programming languages, syntactic abstraction mechanisms, 407
Programming Pearls, 30, 87
Programming Windows with C#, 114
programs generating code while running, 106
progress report for large programs, 51
Project (ERP5), 346–351
coding, 347–351
typical project domain attributes and behavior, 348
relations with Trade, 346
project management, 346
project, defined, 346
pronunciation rules (text preprocessing in Emacspeak), 510
property sheets, 343
DeliveryLine class, 344
protein-coding genes, 188
prototypal nature, JavaScript, 131
public key cryptography, 186
Public Key Infrastructure (PKI)
reimposition of authoritarian cultural tendencies, 186
shortcomings of, 164
public keys tied to contacts in the user's address book (Cryptonite), 167
public-key cryptography, 163
key authentication, 164
punctuation characters, repeated strings (text preprocessing in Emacspeak), 510
punctuations setting (Emacspeak), 512
put-text-property function, advice implementation, 508

Copyrighted material

PyArrayIterObject structure, 312
PyDict_SetItem( ) function, 300
PyDictEntry structure, pointers in, 296
PyDictObject structure, 295
ma_lookup field, pointer to function for looking up keys, 297
special-case optimization for small hashes, 297
PyIntObject type, 298
PyObject structure, 298
PyStringMap class (Jython), 297
PyStringObject type, 298
Python
delta debugging algorithm implementation, 471
dictionaries, 293–301
basic operations on, 293
C implementation of Python, 294
collisions, 298
iterations and dynamic changes, 300
keys and values of different data types in single dictionary, 294
keys not ordered, 294
passing keyword arguments to a function, 294
PyDictObject structure, 295
representing module contents, 294
resizing, 299
special-case optimization for small hashes, 297
special-casing, when it's worthwhile, 297
download site, 293
ERP5, 339
document structure is implemented as portal class, 340
iterators, 305
NumPy, 303–318
iterator design, 307–313
iterator interface, 313
iterator origins, 307
iterator use, 314–318
key challenges in N-dimensional array operations, 304
memory models for N-dimensional array, 305
python command (Mac OS and Linux), 293

Q

Quicksort program, 29–40
aphorisms about beauty, 39
ideas for refining code, 39
paring down code while increasing function, 31–36
average comparisons as pseudocode, 33
calculation with code moved out of loop, 34
calculation with dynamic programming, 34
calculation with inner loop removed, 35
calculation with symmetry, 34
counting comparisons used in array sort, 31
final version of calculation, 36
increment moved out of loop, 32
inserting element into binary search tree, 38
skeleton implemented as a function, 33
skeleton reduced to counting, 32
skeleton with single size argument, 33
techniques used, mathematical analysis of, 37
techniques used, summary of, 37
quicksort( ) function, 30
quote form (syntax-case), 415
exp-quote procedure, 421

R

Rails, pursuit of brevity and DRY, 479
Rake (build tool), 480
Rakefile (example), simplicity of Ruby code, 480
Raman, T. V., 503–525
random numbers
generating binomially distributed random integers, 160
random-number generator facility of NumPy, 317
ranking search results, 56
raster operations, 107–112
BitBlt and StretchBlt functions, Windows 1.0, 108
Reactive_Logging_Server
evaluating, 443
main program that uses the socket API, 442
Reactive_Logging_Server class, 441
Reactor pattern, 434
readability of code, 477–481
Recall replication framework, 175
records (faulty), skipping in MapReduce implementation, 380
recursion, 6
contribution to small, clean, elegant code, 9
for loops looping over N-dimensional arrays, 304
LU factorization, 240–243
Recursive Descent, 129
Recursive LU Algorithm, basic steps, 242
redundancy, eliminating in code, 479
reentrant functions, 227

Copyrighted material

re-execution (transactions in STM), 394
re-expression and symmetry in programming, 34, 37
ref( ) function (Perl), 208
references (variable), capture by a variable binding, 408–411
regexps (see regular expressions)
regular expression matcher
  implementation, 3
  reasons for compactness of code, 8
  termination conditions, 6
regular expressions, 1–9, 43–46
  for input validity in Cryptonite mail system, 179
  mapping into actions with lex, 288
  notation, 2
  program for printing article-fetch lines (example), 43–46
regulatory DNA, 188
regulatory protein bound to specific site of the DNA, 206
relational databases
  in eLocutor, 486
    predictor databases, 493
  ERP5, 341
  Gene Sorter and, 223
relationship managers (ERP5), 345
reliability of CIP, 325
  streamer service, 328–336
    logging, 329–334
Replace feature (eLocutor), 494
replication, Cryptonite mail store, 175, 183
Replication::Recall module (Perl), 175
Representational State Transfer (see REST)
representations (syntax-case expansion algorithm), 414
reserved words, 139
resources (ERP5), 341
REST (Representational State Transfer), 452–462
  exchanging data using e-business protocols, 457–462
  exposing services to external clients, 452–455
    defining service interface, 453–455
  routing the service using factory pattern, 456–457
restrictions built into code, causing difficult-to-find errors, 259
retrieval of data, 230
return statement (JavaScript), 143
reuse of code
  ERP5, 351
  promoted by good design and clear, concise code, 261
reusing data to reduce memory traffic, 239
reverse web link graph, constructing with MapReduce, 376
revision number, 12
right associative operators (JavaScript), 136
right mouse button as single binary input for eLocutor, 487
RISC computers
  with cache hierarchies, 230
    effects on matrix algorithms, 231
  fundamental instructions, 147
  population count
    basic methods, 148
river meandering, model of, 547
RNA sequences, 188
RNA, messenger RNA (mRNA), 218
robustness
  CIP (Collaborative Information Portal), 336–337
    dynamic reconfiguration, 337
    hot swapping, 337
  (see also debugging; testing)
Rosettanet e-business protocol, 453
  exchanging requests and responses using, 457–462
  web site, 457
RowFixture class (Java), 77
RPG legacy systems, integrating with modern client/server software, 460
RSS and Atom feeds, 522
rsync's rolling checksum algorithm, 11
Ruby programming language, 43–49
  array implementation, 52
  beautiful code support
    brevity (Hello World example), 478
    brevity and DRY, 479
    familiarity, 479
    simplicity, 480
  counting article fetches, 47
  delimiting code blocks, 48
  optimizing program that reports most popular articles, 49
  stripped-down design, 46
runtime environment (exp procedure), 419

S

same-marks? predicate, 418
Santa Claus problem (concurrent program using STM), 396–403
Savoia, Alberto, 85–103
Sawzall language for logs analysis, 382
scalability
  of beautiful code, 259
  CIP, 325
Scalable Vector Graphics (SVG) images, 212

Copyrighted material

ScaLAPACK
LU factorization, 243–245
PBLAS (Parallel BLAS), 245–247
Scheme language
expanding expressions containing macros into expressions in the core language, 424
primitives for converting strings to and from symbols, 415
quasiquote syntax for creating list structure, 415
syntax-case, 411
transformation of or form into let and if, 408
Schmidt, Douglas C., 429–450
scientific method, applied to program failures, 466
scope, 138–140
functions, 143
new scope for a function or a block, 140
OO framework, 434
scripting language (MATLAB), 233
scrolling in eLocutor, 498
searches, 41–57
binary search, 52
advantages and disadvantages, 54
content-addressable storage, 46–49
emacspeak-websearch module for task-oriented search, 517–520
escaping the loop, 54
optimizing program that reports most popular articles, 49
postings, 55
ranking results, 56
regular expressions, 43–46
using in program that prints article-fetch lines, 43–46
time involved in running and programming, 41
web searches, 56
weblog data, 42
writing search algorithm, 50
secure communications, 161–186
complexity of secure messaging, untangling, 163
Cryptonite mail system, 162–184
hacking the civilization, 185
privacy protection for individual rights, 185
Seiwald, Christopher, 527–537
self-evaluating? predicate, 417
sequential concurrency models, implemented in logging servers, 439–444
serialization, Cryptonite messages, 170
Service class (Cryptonite), 170
Service class (example), 454
service-oriented architecture (see SOA)
services
exposing to external clients, 452–455
in multitiered service-oriented architecture, 320
Session class (Java), 78
session key, Gene Sorter, 220
sets, computing size for sets represented by bit strings, 158
s-expression, 412
conversion to/from syntax object using datum->syntax, 424
representing a quote form, 421
representing an if form, 421
SGBSV routine (LAPACK library), 255–260
implementation details, 261
shadow folder for mailbox messages, 176
shadowed folder for mailbox messages, 176
shards, 56
sharpness filter, 113
shift right immediates instruction, 151
short-circuiting logical operators (JavaScript), 136
shortest match, 5
side effects
in Haskell, 389–391
being explicit about, 391
minimizing, 227
sideways sum (see population count)
Simple Object Access Protocol (SOAP), 452
SimpleItem property sheet, 343
simplicity in code, 261
Ruby programming language, 480
Simplified JavaScript, 130
single binary input for eLocutor, 487
single-binding let forms, transforming with exp-let procedure, 422
single-binding letrec-syntax forms, 422
single-chip multi-core machines, 231
skipping bad records (MapReduce), 380
slicing, 304
slopes
calculating, 542
comparing to test collinearity, 544–545
measurement with respect to the y-axis, 543
small pieces of code, practicing with, 40
smoke tests, 91
SMP (symmetric multiprocessing) machines, 231
SOA (service-oriented architecture)
loose coupling of services with client applications, 324
three-tiered, CIP implementation, 322
SOAP, using REST over, 452
SOCK_Acceptor type, 443

Copyrighted material

software
  bugs as "spoonful of sewage in the barrel of wine", 353
  correctness and purity of, 353
  operated with a single button, 483–501
software engineering principles, 368
software systems, 40
Software Transactional Memory (see STM)
Solaris
  development of critical kernel subsystem, 354
  interaction between kernel memory allocator and Zettabyte Filesystem (ZFS), 356
  priority inheritance for kernel synchronization primitives, 356
  user-level priority inheritance, debugging, 360–366
sorting
  distributed, performing with MapReduce, 376
  MapReduce, ordering guarantees, 380
  sort function, 31
  (see also Quicksort program)
source_project base category, 347
  retrieving tasks of a project, 348
sparse matrices, 230
specifications for code, incompleteness of, 260
speech delivery in eLocutor, 499
speech output, styling with Aural CSS (ACSS), 511–516
speech servers, 504
speech-enabling Emacs, 505–516
  advice tutorial, 507
  first-cut implementation, 505
  generating rich auditory output, 507–511
  implementing event queue in speech server, 505
spin lock (thread lock) in Solaris, 358
Spinellis, Diomidis, 279–291
splitting text into appropriate clauses (Emacspeak), 510
spoken output, producing, 504
"spoonful of sewage in the barrel of wine," software bugs as, 353
SQL
  code to fetch data for Gene Sorter association columns, 224
  mail storage backend (Cryptonite), 175
SQL database, integration with cookie and hidden variable mechanisms, 220
sqrt returning an irrational result, 546
standards-based applications, beauty of, 324
start position (genomic features), 190
stateful session beans, 323
  CIP streamer service, 326
stateless session beans, 323
  CIP streamer service provider, 326
statements (JavaScript), 140–143
std (statement denotation) method (JavaScript tokens), 140
Stein, Lincoln, 187–215
STM (Software Transactional Memory), 386, 388–405
  implementing transactional memory, 394
  Santa Claus problem (example program), 396–403
  side effects and I/O in Haskell, 388–391
  summary of basic operations, 396
  transactions in Haskell, 392–393
STM actions, 392
  forming a list and combining with orElse, 403
  giving functions STM types wherever possible, 400
storage and retrieval of data, 230
Store class (Java), 79
Strategy pattern, 437
streamer service (CIP), 325–336
  functionality, 325
  reliability, 328–336
    logging, 329–334
  service architecture, 326
streaminess (Subversion interface), 23
stress tests, 369
StretchBlt function (Windows 1.0), 106
  raster operations, 108
string-specialized dictionary type, 297
strip procedure, 415
structures
  base structure field defining pointer type, 272
  dictionary structures no longer in use, 300
  holding packed function arguments, 288
  inheritance and manipulation on Linux kernel, 269–272
  in multithreaded programs, reference counting, 273–276
  NumPy iterator, 312
  PyDictObject, 295
  PyObject, 298
  struct_inode and struct_dentry, putting in kernel caches, 277
subfeatures and sub-subfeatures (genomic features), 190
subroutines, logical division of tasks into, 265
substitutions, 413
  appearing in an identifier's wrap, determining associated label, 418
  mapping a symbolic name and list of marks to a label, 415
subsystem, same design assumptions for memory usage as main system, 261
subtracts instruction, 151
Subversion, delta editor, 11–28
summing factor technique, 37

Copyrighted material

SVG (Scalable Vector Graphics) images, 212
svn_delta_editor_t (see delta editor)
svn_error_t (Subversion error type), 17
svnsync functionality (Subversion), 27
switch statements (Java), 68
problems with larger groups of cases, 70
symbol table (JavaScript parser), 131
symmetric multiprocessing (SMP) machines, 231
symmetry, exploiting in tuning Quicksort loop, 34
synchronization mechanisms
ACE wrapper facades, 436
associating concurrency strategy with, 437
synchronized methods, 386
synchronous event demultiplexing APIs (operating systems), 441
SyncML protocol, 341
syntactic abstraction, 407–428
syntax form, 411
use by expander, 420
syntax objects, 414
conversion of s-expressions to/from, using datum->syntax, 424
nonatomic structure, determining, 416
parsing and constructing, 423
stripping, 415
syntax->datum procedure, 413
syntax-car operator, 423
syntax-case
expander example, 425–427
expansion algorithm, 413–425
comparing identifiers, 423
conversions, 424
core transformers, 421–423
creating wraps, 417
expander, 419–421
identifier resolution, 418
manipulating environments, 418
parsing and constructing syntax objects, 423
producing expander output, 415
representations, 414
starting expansion, 424
stripping syntax objects, 415
structural predicates, 416
syntax errors, 416
introduction to, 411–413
datum->syntax procedure, 413
hygiene, bending or breaking, 412
input or output form followed by ellipsis, 412
macro definition, 411
method form, 413
with-syntax form, 413
syntax-case form, 411
use by expander, 420
syntax-case system, 411
syntax-cdr operator, 423
syntax-error, 416
syntax-pair? operator, 423
syntax-rules system, 410
sysfs (Linux), 269
excessive memory usage, 276
rewriting code to put struct inode and struct dentry structures in kernel caches, 277
systematic debugging, 466
further reading, 476

T

table lookup, 68
using in population count, 149
tallying, history of, 50
Task Report workflow, 350
task reports (ERP5), 347
taskReport_notifyAssignee script, 350
tasks (ERP5), 347
associated with a project, 347
retrieving, 348
task predecessors, 349
Task workflow, 349
TCL language, implementation of Emacspeak speech servers, 504
template identifier, 412
Template Method pattern, 432
in Logging_Server base class run( ) method, 450
structure and application to logging server, 435
template specifying output in define-syntax form, 412
Templates (in eLocutor), 494
term vector per host, constructing with MapReduce, 376
termination conditions (regular expression matcher), 6
ternary operator (?:) in JavaScript, 135
ternary raster operation, 108
testing, 85–103
beauty in tests, 86
binary search, 87, 90–102
boundary value tests, 92–95
performance, 101
random testing, 95–100
smoke test, 91
continuous, 467
Cryptonite mail system, 172
FIT (Framework for Integrated Test), 75–84
importance of test programs, 260
JUnit testing framework, 89
text delta (Subversion), 17
text preprocessing before sending to speech server (Emacspeak tts-speak), 510

Copyrighted material

text searches, 55
text, splitting into clauses for speech synthesis, 506
Text::Template module (Perl), 173
Thompson, Ken, 1
thread lock, 358
three-dimensional arrays, 303
time management (MER mission), 321
tokens
JavaScript, 132
arity, 132
precedence decisions, 133
std method, 140
token-based C preprocessor macros, 408
Top Down Operator Precedence, 129
top mark, 416
Track class (Bio::Graphics), 198
tracks (Bio::Graphics module), 193–195
configuration options, 196
positions and relative strengths of DNA/protein binding site, 206
subclass of Glyph, 199
Trade BT, 346
trailing 0s in a word, computing number of, 159
transaction log, 394
transactional memory, 386, 405
(see also STM)
transactions (in Haskell), 392–393
transformers, 411
core, 421–423
transposition operator ('), 233
transposons, 218
tree delta (Subversion editor), processing each piece upon creation, 23
tree structure presenting options in eLocutor, 487, 488
dynamic repopulation, 491
tree transformation, version control and, 12–15
triangle inequality, testing collinearity by, 545
triangle-collinear function, 546
tts-format-text-and-speak function (Emacspeak), 510
tts-speak function, 507
tts-speak module (Emacspeak), 510
tts-speak-using-voice function, 511
turnstile_block( ) function, 354
turnstile_pi_waive( ) function, 366
turnstiles (Solaris), 354
priority inversion, 355
two-dimensional arrays, 303
type (bindings), 415
type checking for pointers, 272
type signature (Haskell), 389
TypeAdapter class (Java), 77
typing in eLocutor, 493, 497
speeding up, 485

U

UBM (Unified Business Model), 341
udev tool (Linux), 272
ufs filesystem (BSD 4.2), 283
umapfs filesystem, 282
interposing over NFS implementation, 285
understandable code, keys to, 225–228
brevity and simplicity of code, 261
documentation, 227
effect of indirection on comprehension of code, 291
limits of human memory, 225
minimizing side effects, 227
single, consistent style for names, 227
using scope as local as possible, 227
well-chosen names, 226
unhygienic macro expansion, 410
Unicode code points in XML, 60
Unified Business Model (UBM), 341
Unix operating system
input and output sources, uniform treatment of, 289
read system call variants, 289
socket API, 429
Unix-related operating systems, use of function pointers, 282
unordered hash (Perl), 195
unsigned modulus instruction, 152
URIs (Uniform Resource Identifiers), caching after verifying for XML input, 73
URL templates (Emacspeak), 520
Google News searches via Atom feeds, 522
retrieving weather for a city or state, 522
usability considerations in public-key cryptosystems, 166
USB devices in Linux sysfs, 269
USB sticks, FAT-32 filesystem, 279
usb_interface structure (Linux), 271
pointer, getting with container_of macro, 272
structures embedded within other structures, 276
user key for Gene Sorter, 220
user-defined control structures (in Haskell), 391
user-level priority inheritance bug, Solaris, 354–368

V

validation
transaction log, 394
user input, Cryptonite mail system, 179
XML, 59
validity (XML), 60
value (bindings), 415

Copyrighted material

values
Python dictionaries
different data types in a single dictionary, 294
searching for, 46
var statement (JavaScript), 141
variabilities in OO framework analysis, 434
variable capture problems, 408–411
solving with hygienic macro expansion, 409
variable names, Perl, 195
variables
CGI, 220
defining in current block (JavaScript), 141
naming, understanding purpose from the name, 226
reading/writing a mutable variable, side effect in Haskell, 390
scope in JavaScript, 138–140
VB (Visual Basic), 485
TreeView control, 488
indexing feature speeding up retrieval, 492
vector machines, 230
vectorization of linear algebra algorithms, 237
vector of I/O requests, Unix read system calls, 289
vectorization and parallelism, algorithmic approach for exploiting, 230
vector-vector operations
denoted by Level-1, 235
expressing matrix algorithms as, 231
verifiers, XML (see XML verifiers)
version-control system (see Subversion, delta editor)
vertical line, y-intercept, 542
virtual evaluation stack (in IL), 120
virtual machines, 290
vnode call interface, FreeBSD version 6.1 implementation, 288
vnodeop_desc structure, 286
voice overlays (Emacspeak), 512
voice-lock (Emacspeak), 508
voice-monotone, ACSS setting corresponding to, 512
vop_generic_args structure (FreeBSD), 286
vop_vector structure (FreeBSD), 280
filesystem layering support, 282
pointer to a bypass function, 284
pointer to vop_vector structure of underlying filesystem layer, 284
populating bypass and default fields, filesystem choices resulting from, 284

W

waking up threads in Solaris, 354
Warren, Henry S., Jr., 147–160
web applications, use of Bio::Graphics output, 191, 210
web link graphs, constructing with MapReduce, 376
web of trust
key authentication stronger than PKI, 186
key management interface in Cryptonite, 177
visibility of information in Cryptonite Key Ring view, 166
web page for this book, xxi
web searches, 55, 56
web services
communication between client applications and middleware services, 324
use by CIP streamer service for client requests and responses, 326
Web Services architecture, decision points for choosing, 452
web site for this book, 382
web.xml file for servlet providing user interface, 454
web-based genome browsers based on Bio::Graphics, 211
weblogs, searching, 42
web-oriented tools in Emacspeak, 516–522
basic HTML with Emacs W3 and ACSS, 517
feed readers, 522
Web command line and URL templates, 520
websearch module for task-oriented search, 517–520
webs of trust, 164
well-formedness (XML), 60
Wheeler, David, 290
while loop, 4
while statement (JavaScript), 142
wildcards
in regular expressions, 1
shell, modifying regular expressions to resemble, 8
window handler (Subversion), 17
Windows Forms code (ImageClip program), 114
Windows operating systems
native filesystem, 279
threading API, 429
Wingerd, Laura, 527–537
with-syntax form, 413

Copyrighted material

word completion in eLocutor, 485, 493
word count program (example), 371–374
  C++ implementation, 382
  division of problem into Map and Reduce
    functions, 374
  parallelized, 372
  parallelized word count program
    with partitioned processors, 373
    with partitioned storage, 372
word groupings in eLocutor, 486
words, common (eLocutor), 496
workflows (ERP5)
  implementing task behavior, 349
  Task Report, 350
working copy (Subversion), 15
Wrapper Facade pattern, 432, 436
wrapper facades, use in sequential logging
    server implementation, 444
wraps, 413
  creating, 417
  stripping, 415

X

XML
  assembling XML response, 462
  parsing data using XPath, 458–462
  request/response data via HTTP POST, 453
  Version 1.0 specification, 61
XML technologies, ERP5 reliance on, 341
XML verifiers, 59–74
  first optimization, Version 3, 64
  fourth optimization, caching namespace
    URIs after verification, 72
  name character verification, version 1, 62
  role of validation, 59
  second optimization, Version 4, 66
  third optimization, lookup table, 68–72
  XML-based named character verification,
    Version 2, 63
XMLObject class, 343

Y

yacc (parser generator), input of language
    grammar and production rules, 288
Yahoo! Maps, accessing directions with
    emacspeak-websearch tool, 518
y-intercept of a vertical line, 542

Z

Zeller, Andreas, 463–476
ZFS (Zettabyte Filesystem), 356
Zimmermann, Phil, 164
ZODB, 340
Zope platform, 339, 342–346
  CMF (Content Management
    Framework), 340, 342
  key components used by ERP5, 340
ZPT (Zope Page Templates), 340

Copyrighted material

Programming/Software Engineering

Beautiful Code

This unique and insightful book is a collection of master classes in software design. In each chapter, today's leading programmers walk through elegant solutions to hard problems, and explain what makes those solutions so appealing.

This is not simply another design patterns book, or another software engineering treatise on the right and wrong ways to do things. Instead, it gives you the chance to look over the shoulder of some superb software designers and see the world through their eyes.

Thirty-eight master coders think aloud as they work through a project's architecture, the tradeoffs made in its construction, and those moments when it was important to break rules.

The book includes contributions from:

Brian Kernighan
Karl Fogel
Jon Bentley
Tim Bray
Elliotte Rusty Harold
Michael Feathers
Alberto Savoia
Charles Petzold
Douglas Crockford
Henry S. Warren, Jr.
Ashish Gulhati
Lincoln Stein
Jim Kent
Jack Dongarra and Piotr Luszczek
Adam Kolawa
Greg Kroah-Hartman
Diomidis Spinellis
Andrew Kuchling
Travis E. Oliphant
Ronald Mak
Rogerio Atem de Carvalho and Rafael Monnerat
Bryan Cantrill
Jeff Dean and Sanjay Ghemawat
Simon Peyton Jones
R. Kent Dybvig
William R. Otte and Douglas C. Schmidt
Andrew Patzer
Andreas Zeller
Yukihiro Matsumoto
Arun Mehta
T. V. Raman
Laura Wingerd and Christopher Seiwald
Brian Hayes

All author royalties will be donated to Amnesty International.

O'REILLY®
www.oreilly.com

US $44.99 CAN $58.99
ISBN-10: 0-596-51004-7
ISBN-13: 978-0-596-51004-6

Safari® BOOKS ONLINE ENABLED
Includes FREE 45-Day Online Edition

Copyrighted material