Mastering Regular Expressions - Table of Contents

Mastering Regular Expressions

Table of Contents

Tables

Preface

1 Introduction to Regular Expressions

2 Extended Introductory Examples

3 Overview of Regular Expression Features and Flavors

4 The Mechanics of Expression Processing

5 Crafting a Regular Expression

6 Tool-Specific Information

7 Perl Regular Expressions

A Online Information

B Email Regex Program

Index

Mastering Regular Expressions

Powerful Techniques for Perl and Other Tools

Jeffrey E.F. Friedl

O'REILLY™
Cambridge • Köln • Paris • Sebastopol • Tokyo

Mastering Regular Expressions
by Jeffrey E.F. Friedl

Copyright © 1997 O'Reilly & Associates, Inc. All rights reserved.
Printed in the United States of America.

Published by O'Reilly & Associates, Inc., 101 Morris Street, Sebastopol, CA 95472.

Editor: Andy Oram
Production Editor: Jeffrey Friedl

Printing History:

January 1997:	First Edition.
March 1997:	Second printing; Minor corrections.
May 1997:	Third printing; Minor corrections.
July 1997:	Fourth printing; Minor corrections.
November 1997:	Fifth printing; Minor corrections.
August 1998:	Sixth printing; Minor corrections.
December 1998:	Seventh printing; Minor corrections.

Nutshell Handbook and the Nutshell Handbook logo are registered trademarks and The Java Series is a trademark of O'Reilly & Associates, 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 & Associates, 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 assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

Table of Contents

Preface xv

1: Introduction to Regular Expressions 1

Solving Real Problems 2

Regular Expressions as a Language 4

The Filename Analogy 4

The Language Analogy 5

The Regular-Expression Frame of Mind 6

Searching Text Files: Egrep 7

Egrep Metacharacters 8

Start and End of the Line 8

Character Classes 9

Matching Any Character—Dot 11

Alternation 12

Word Boundaries 14

In a Nutshell 15

- Optional Items 16
- Other Quantifiers: Repetition 17
- Ignoring Differences in Capitalization 18
- Parentheses and Backreferences 19
- The Great Escape 20

Expanding the Foundation 21

- Linguistic Diversification 21
- The Goal of a Regular Expression 21
- A Few More Examples 22

        Regular Expression Nomenclature 24

        Improving on the Status Quo 26

        Summary 28

    Personal Glimpses 30

2: Extended Introductory Examples 31

    About the Examples 32

        A Short Introduction to Perl 33

    Matching Text with Regular Expressions 34

        Toward a More Real-World Example 36

        Side Effects of a Successful Match 36

        Intertwined Regular Expressions 39

        Intermission 43

    Modifying Text with Regular Expressions 45

        Automated Editing 47

        A Small Mail Utility 48

        That Doubled-Word Thing 54

3: Overview of Regular Expression Features and Flavors. 59

A Casual Stroll Across the Regex Landscape 60

  The World According to Grep 60

  The Times They Are a Changin' 61

At a Glance 63

  POSIX 64

Care and Handling of Regular Expressions 66

  Identifying a Regex 66

  Doing Something with the Matched Text 67

  Other Examples 67

  Care and Handling: Summary 70

Engines and Chrome Finish 70

  Chrome and Appearances 71

  Engines and Drivers 71

Common Metacharacters 71

  Character Shorthands 72

  Strings as Regular Expression 75

  Class Shorthands, Dot, and Character Classes 77

  Anchoring 81

Grouping and Retrieving 83

Quantifiers 83

Alternation 84

Guide to the Advanced Chapters 85

Tool-Specific Information 85

4: The Mechanics of Expression Processing 87

Start Your Engines! 87

Two Kinds of Engines 87

New Standards 88

Regex Engine Types 88

From the Department of Redundancy Department 90

Match Basics 90

About the Examples 91

Rule 1: The Earliest Match Wins 91

The "Transmission" and the Bump-Along 92

Engine Pieces and Parts 93

Rule 2: Some Metacharacters Are Greedy 94

Regex-Directed vs. Text-Directed 99

NFA Engine: Regex-Directed 99

DFA Engine: Text-Directed 100

The Mysteries of Life Revealed 101

Backtracking 102

A Really Crummy Analogy 102

Two Important Points on Backtracking 103

Saved States 104

Backtracking and Greediness 106

More About Greediness 108

Problems of Greediness 108

Multi-Character "Quotes" 109

Laziness? 110

Greediness Always Favors a Match 110

Is Alternation Greedy? 112

Uses for Non-Greedy Alternation 113

Greedy Alternation in Perspective 114

Character Classes vs. Alternation 115

NFA, DFA, and POSIX 115

"The Longest-Leftmost" 115

POSIX and the Longest-Leftmost Rule 116

Speed and Efficiency 118

DFA and NFA in Comparison 118

Practical Regex Techniques 121

Contributing Factors 121

Be Specific 122

Difficulties and Impossibilities 125

Watching Out for Unwanted Matches. 127

Matching Delimited Text 129

Knowing Your Data and Making Assumptions 132

Additional Greedy Examples 132

Summary 136

Match Mechanics Summary 136

Some Practical Effects of Match Mechanics 137

5: Crafting a Regular Expression 139

A Sobering Example 140

A Simple Change-Placing Your Best Foot Forward 141

More Advanced-Localizing the Greediness 141

Reality Check 144

A Global View of Backtracking 145

- More Work for a POSIX NFA 147
- Work Required During a Non-Match. 147
- Being More Specific 147
- Alternation Can Be Expensive 148
- A Strong Lead 149
- The Impact of Parentheses 150

Internal Optimization 154

- First-Character Discrimination 154
- Fixed-String Check 155
- Simple Repetition 155
- Needless Small Quantifiers 156
- Length Cognizance 157
- Match Cognizance 157
- Need Cognizance 157
- String/Line Anchors 158
- Compile Caching 158

Testing the Engine Type 160

- Basic NFA vs. DFA Testing 160

  Traditional NFA vs. POSIXNFA Testing 161

Unrolling the Loop 162

  Method 1: Building a Regex From Past Experiences 162

    The Real "Unrolling the Loop" Pattern. 164
    Method 2: A Top-Down View 166
    Method 3: A Quoted Internet Hostname 167
    Observations 168
  Unrolling C Comments 168
    Regex Headaches 169
    A Naive View 169
    Unrolling the C Loop 171
  The Freeflowing Regex 173
    A Helping Hand to Guide the Match. 173
    A Well-Guided Regex is a Fast Regex. 174
    Wrapup 176
  Think! 177
    The Many Twists and Turns of Optimizations 177
6: Tool-Specific Information 181
  Questions You Should Be Asking 181
    Something as Simple as Grep 181

In This Chapter 182

Awk 183

Differences Among Awk Regex Flavors 184

Awk Regex Functions and Operators 187

Tcl 188

Tcl Regex Operands 189

Using Tcl Regular Expressions 190

Tcl Regex Optimizations 192

GNU Emacs 192

Emacs Strings as Regular Expressions 193

Emacs's Regex Flavor 193

Emacs Match Results 196

Benchmarking in Emacs 197

Emacs Regex Optimizations 197

7: Perl Regular Expressions 199

The Perl Way 201

Regular Expressions as a Language Component 202

Perl's Greatest Strength 202

Perl's Greatest Weakness 203

- An Introductory Example: Parsing CSV Text 204
- Regular Expressions and The Perl Way 207
- Perl Unleashed 208

Regex-Related Perlisms 210

- Expression Context 210
- Dynamic Scope and Regex Match Effects 211
- Special Variables Modified by a Match 217
- "Doublequotish Processing" and Variable Interpolation 219

Perl's Regex Flavor 225

- Quantifiers-Greedy and Lazy 225
- Grouping 227
- String Anchors 232
- Multi-Match Anchor 236
- Word Anchors 240
- Convenient Shorthands and Other Notations 241
- Character Classes 243
- Modification with \Q and Friends: True Lies 245

The Match Operator 246

    Match-Operand Delimiters 247

    Match Modifiers 249

    Specifying the Match Target Operand 250

    Other Side Effects of the Match Operator 251

    Match Operator Return Value 252

    Outside Influences on the Match Operator 254

The Substitution Operator 255

    The Replacement Operand 255

    The /e Modifier 257

    Context and Return Value 258

    Using /g with a Regex That Can Match Nothingness 259

The Split Operator 259

    Basic Split 259

    Advanced Split 261

    Advanced Split's Match Operand 262

    Scalar-Context Split 264

    Split's Match Operand with Capturing Parentheses 264

Perl Efficiency Issues 265

"There's More Than One Way to Do It" 266

Regex Compilation, the /o Modifier, and Efficiency 268

Unsociable $& and Friends 273

  - The Efficiency Penalty of the /i Modifier 278
  - Substitution Efficiency Concerns 281
  - Benchmarking 284
  - Regex Debugging Information 285
  - The Study Function 287
- Putting It All Together 290
  - Stripping Leading and Trailing Whitespace 290
  - Adding Commas to a Number 291
  - Removing C Comments 292
  - Matching an Email Address 294
- Final Comments 304
  - Notes for Perl4 305

A Online Information 309

BEmail Regex Program 313

Tables

1-1 Summary of Metacharacters Seen So Far 15

1-2 Summary of Quantifier "Repetition Metacharacters" 18

1-3 Egrep Metacharacter Summary 29

3-1 A (Very) Superficial Look at the Flavor of a Few Common Tools 63

3-2 Overview of POSIX Regex Flavors 64

3-3 A Few Utilities and Some of the Shorthand Metacharacters They Provide 73

3-4 String/Line Anchors, and Other Newline-Related Issues 82

4-1 Some Tools and Their Regex Engines 90

5-1 Match Efficiency for a Traditional NFA 143

5-2 Unrolling-The-Loop Example Cases 163

5-3 Unrolling-The-Loop Components for C Comments 172

6-1 A Superficial Survey of a Few Common Programs' Flavor 182

6-2 A Comical Look at a Few Greps 183

6-3 A Superficial Look at a Few Awks 184

6-4 Tcl's FA Regex Flavor 189

6-5 GNU Emacs's Search-Related Primitives 193

6-6 GNU Emacs's String Metacharacters 194

6-7 Emacs's NFA Regex Flavor 194

6-8 Emacs Syntax Classes 195

7-1 Overview of Perl's Regular-Expression Language 201

7-2 Overview of Perl's Regex-Related Items 203

7-3 The meaning of local 213

7-4 Perl's Quantifiers (Greedy and Lazy) 225

7-5 Overview of Newline-Related Match Modes 232

7-6 Summary of Anchor and Dot Modes 236

7-7 Regex Shorthands and Special-Character Encodings 241

7-8 String and Regex-Operand Case-Modification Constructs 245

7-9 Examples of m/…/g with a Can-Match-Nothing Regex 250

7-10 Standard Libraries That Are Naughty (That Reference $& and Friends) 278

7-11 Somewhat Formal Description of an Internet Email Address 295

Preface

This book is about a powerful tool called "regular expressions."

Here, you will learn how to use regular expressions to solve problems and get the most out of tools that provide them. Not only that, but much more: this book is about *mastering* regular expressions.

If you use a computer, you can benefit from regular expressions all the time (even if you don't realize it). When accessing World Wide Web search engines, with your editor, word processor, configuration scripts, and system tools, regular expressions are often provided as "power user" options. Languages such as Awk, Elisp, Expect, Perl, Python, and Tcl have regular-expression support built in (regular expressions are the very heart of many programs written in these languages), and regular-expression libraries are available for most other languages. For example, quite soon after Java became available, a regular-expression library was built and made freely available on the Web. Regular expressions are found in editors and programming environments such as *vi,* Delphi, Emacs, Brief, Visual C++, Nisus Writer, and many, many more. Regular expressions are very popular.

There's a good reason that regular expressions are found in so many diverse applications: they are extremely powerful. At a low level, a regular expression describes a chunk of text. You might use it to verify a user's input, or perhaps to sift through large amounts of data. On a higher level, regular expressions allow you to master your data. Control it. Put it to work for you. To master regular expressions is to master your data.

Why I Wrote This Book

You might think that with their wide availability, general popularity, and unparalleled power, regular expressions would be employed to their fullest, wherever found. You might also think that they would be well documented, with introductory tutorials for the novice just starting out, and advanced manuals for the expert desiring that little extra edge.

Sadly, that hasn't been the case. Regular-expression documentation is certainly plentiful, and has been available for a long time. (I read my first regular-expression-related manual back in 1981.) The problem, it seems, is that the documentation has traditionally centered on the "low-level view" that I mentioned a moment ago. You can talk all you want about how paints adhere to canvas, and the science of how colors blend, but this won't make you a great painter. With painting, as with any art, you must touch on the human aspect to really make a statement. Regular expressions, composed of a mixture of symbols and text, might seem to be a cold, scientific enterprise, but I firmly believe they are very much creatures of the right half of the brain. They can be an outlet for creativity, for cunningly brilliant programming, and for the elegant solution.

I'm not talented at anything that most people would call art. I go to karaoke bars in Kyoto a lot, but I make up for the lack of talent simply by being loud. I do, however, feel very artistic when I can devise an elegant solution to a tough problem. In much of my work, regular expressions are often instrumental in developing those elegant solutions. Because it's one of the few outlets for the artist in me, I have developed somewhat of a passion for regular expressions. It is my goal in writing this book to share some of that passion.

Intended Audience

This book will interest anyone who has an opportunity to use regular expressions. In particular, if you don't yet understand the power that regular expressions can provide, you should benefit greatly as a whole new world is opened up to you. Many of the popular cross-platform utilities and languages that are featured in this book are freely available for MacOS, DOS/Windows, Unix, VMS, and more. Appendix A has some pointers on how to obtain many of them.

Anyone who uses GNU Emacs or *vi*, or programs in Perl, Tcl, Python, or Awk, should find a gold mine of detail, hints, tips, and *understanding* that can be put to immediate use. The detail and thoroughness is simply not found anywhere else. Regular expressions are an idea—one that is implemented in various ways by various utilities (many, many more than are specifically presented in this book). If you master the general concept of regular expressions, it's a short step to mastering a particular implementation. This book concentrates on that idea, so most of the knowledge presented here transcend the utilities used in the examples.

How to Read This Book

This book is part tutorial, part reference manual, and part story, depending on when you use it. Readers familiar with regular expressions might feel that they can immediately begin using this book as a detailed reference, flipping directly to the section on their favorite utility. I would like to discourage that.

This Book, as a Story

To get the most out of this book, read it first as a story. I have found that certain habits and ways of thinking can be a great help to reaching a full understanding, but such things are absorbed over pages, not merely memorized from a list. Here's a short quiz: define the word "between" Remember, you can't use the word in its definition! Have you come up with a good definition? No? It's tough! It's lucky that we all know what "between" means because most of us would have a devil of a time trying to explain it to someone that didn't know. It's a simple concept, but it's hard to describe to someone who isn't already familiar with it. To some extent, describing the details of regular expressions can be similar. Regular expressions are not really that complex, but the descriptions can tend to be. I've crafted a story and a way of thinking that begins with Chapter 1, so I hope you begin reading there. Some of the descriptions *are* complex, so don't be alarmed if some of the more detailed sections require a second reading. Experience is 9/10 of the law (or something like that), so it takes time and experience before the overall picture can sink in.

This Book, as a Reference

This book tells a story, but one with many details. Once you've read the story to get the overall picture, this book is also useful as a reference. I've used cross references liberally, and I've worked hard to make the index as useful as possible. (Cross references are often presented as "☞" followed by a page number.) Until you read the full story, its use as a reference makes little sense. Before reading the story, you might look at one of the tables, such as the huge chart on page 182, and think it presents all the relevant information you need to know. But a great deal of background information does not appear in the charts themselves, but rather in the associated story. Once you've read the story, you'll have an appreciation for the issues, what you can remember off the top of your head, and what is important to check up on.

Organization

The seven chapters of this book can be logically divided into roughly three parts, with two additional appendices. Here's a quick overview:

The Introduction

Chapter 1 introduces the concept of regular expressions.
Chapter 2 takes a look at text processing with regular expressions.
Chapter 3 provides an overview of features and utilities, plus a bit of history.

The Details

Chapter 4 explains the details of how regular expressions work.
Chapter 5 discusses ramifications and practical applications of the details.

Tool-Specific Information

Chapter 6 looks at a few tool-specific issues of several common utilities.
Chapter 7 looks at everything to do with regular expressions in Perl.

Appendices

Appendix A tells how to acquire many of the tools mentioned in this book.
Appendix B provides a full listing of a program developed in Chapter 7.

The Introduction

The introduction elevates the absolute novice to "issue-aware" novice. Readers with a fair amount of experience can feel free to skim the early chapters, but I particularly recommend Chapter 3 even for the grizzled expert.

- Chapter 1, *Introduction to Regular Expressions*, is geared toward the complete novice. I introduce the concept of regular expressions using the widely available program *egrep*, and offer my perspective on how to *think* regular expressions, instilling a solid foundation for the advanced concepts in later chapters. Even readers with former experience would do well to skim this first chapter.

- Chapter 2, *Extended Introductory Examples,* looks at real text processing in a programming language that has regular-expression support. The additional examples provide a basis for the detailed discussions of later chapters, and show additional important thought processes behind crafting advanced regular expressions. To provide a feel for how to "speak in regular expressions," this chapter takes a problem requiring an advanced solution and shows ways to solve it using two unrelated regular-expression-wielding tools.

- Chapter 3, *Overview of Regular Expression Features and Flavors,* provides an overview of the wide range of regular expressions commonly found in tools today. Due to their turbulent history, current commonly used regular expression flavors can differ greatly. This chapter also takes a look at a bit of the history and evolution of regular expressions and the programs that use them. The end of this chapter also contains the "Guide to the Advanced Chapters." This guide is your road map to getting the most out of the advanced material that follows.

The Details

Once you have the basics down, it's time to investigate the *how* and the *why.* Like the "teach a man to fish" parable, truly understanding the issues will allow you to apply that knowledge whenever and wherever regular expressions are found. That true understanding begins in:

- Chapter 4, *The Mechanics of Expression Processing,* ratchets up the pace several notches and begins the central core of this book. It looks at the important inner workings of how regular expression engines really work from a *practical* point of view. Understanding the details of how a regular expression is used goes a very long way toward allowing you to master them.

- Chapter 5, *Crafting a Regular Expression,* looks at the real-life ramifications of the regular-expression engine implemented in popular tools such as Perl, *sed, grep,* Tcl, Python, Expect, Emacs, and more. This chapter puts information detailed in Chapter 4 to use for exploiting an engine's strengths and stepping around its weaknesses.

Tool-Specific Information

Once the lessons of Chapters 4 and 5 are under your belt, there is usually little to say about specific implementations. However, I've devoted an entire chapter to one very notable exception, the Perl language. But with any implementation, there *are* differences and other important issues that should be considered.

- Chapter 6, *Tool-Specific Information,* discusses tool-specific concerns, highlighting many of the characteristics that vary from implementation to implementation. As examples, *awk,* Tcl, and GNU Emacs are examined in more depth than in the general chapters.

- Chapter 7, *Perl Regular Expressions,* closely examines regular expressions in Perl, arguably the most popular regular-expression-laden programming language in popular use today. There are only three operators related to regular expressions, but the myriad of options and special cases provides an extremely rich set of programming options—and pitfalls. The very richness that allows the programmer to move quickly from concept to program can be a minefield for the uninitiated. This detailed chapter will clear a path.

Typographical Conventions

When doing (or talking about) detailed and complex text processing, being precise is important. The mere addition or subtraction of a space can make a world of difference, so I use the following special conventions:

- A regular expression generally appears like ⌈`this`⌋. Notice the thin corners which flag "this is a regular expression." Literal text (such as that being searched) generally appears like `'this'`. At times, I'll feel free to leave off the thin corners or quotes when obviously unambiguous. Also, code snippets and screen shots are always presented in their natural state, so the quotes and corners are not used in such cases.

- Without special presentation, it is virtually impossible to know how many spaces are between the letters in `"a    b"`, so when spaces appear in regular expressions and selected literal text, they will be presented with the `'•'` symbol. This way, it will be clear that there are exactly four spaces in `'a••••b'`. I also use visual tab and newline characters. Here's a summary of the three:

•	a space character
TAB	a tab character
NL	a newline character

- At times, I use underlining, or shade the background to highlight parts of literal text or a regular expression. For example:

- Because ⌈`cat`⌋ matches `'It•indicates•your•cat•is…'` instead of the word `'cat'`, we realize. . .

  In this case, the underline shows where in the text the expression actually matched. Another example would be:

To make this useful, we can wrap ⌜`Subject|Date`⌟ with parentheses, and append a colon and a space. This yields ⌜`(Subject|Date):•`⌟

In this case, the underlines highlight what has just been added to an expression under discussion.

- I use a visually distinct ellipses within literal text and regular expressions. For example `[…]` represents a set of square brackets with unspecified contents, while `[. . .]` would be a set containing three periods.

Exercises

Occasionally, particularly in the early chapters, I'll pose a question to highlight the importance of the concept under discussion. They're not there just to take up space; I really do want you to try them before continuing. Please. So as to not to dilute their importance, I've sprinkled only a few throughout the entire book. They also serve as checkpoints: if they take more than a few moments, it's probably best to go over the relevant section again before continuing on.

To help entice you to actually think about these questions as you read them, I've made checking the answers a breeze: just turn the page. Answers to questions marked with ❖ are always found by turning just one page. This way, they're out of sight while you think about the answer, but are within easy reach.

Personal Comments and Acknowledgments

My Mom once told me that she couldn't believe that she ever married Dad. When they got married, she said, they *thought* that they loved each other. It was nothing, she continued, compared with the depth of what they now share, thirty-something years later. It's just as well that they didn't know how much better it could get, for had they known, they might have considered what they had to be too little to chance the future on.

The analogy may be a bit melodramatic, but several years ago, I *thought* I understood regular expressions. I'd used them for years, programming with *awk*, *sed*, and Perl, and had recently written a rather full regular-expression package that fully supported Japanese text. I didn't know any of the theory behind it—I just sort of reasoned it out myself. Still, my knowledge seemed to be enough to make me one of the local experts in the Perl newsgroup. I passed along some of my posts to a friend, Jack Halpern （春遍雀來）, who was in the process of learning Perl. He often suggested that I write a book, but I never seriously considered it. Jack has written over a dozen books himself (in various languages, no less), and when someone like that suggests you write a book, it's somewhat akin to Carl Lewis telling you to just jump far. Yeah, sure, easy for you to say!

Then, toward the end of June, 1994, a mutual friend, Ken Lunde （小林剣）, also suggested I write a book. Ken is also an author (O'Reilly & Associates' *Understanding Japanese Information Processing*), and the connection to O'Reilly was too much to pass by. I was introduced to Andy Oram, who became my editor, and the project took off under his guidance.

I soon learned just how much I didn't know.

Knowing I would have to write about more than the little world of the tools that I happened to use, I thought I would spend a bit of time to investigate their wider use. This began what turned out to be an odyssey that consumed the better part of two years. Just to understand the characteristics of a regular-expression flavor, I ended up creating a test suite implemented in a 60,000-line shell script. I tested dozens and dozens of programs. I reported numerous bugs that the suite discovered (many of which have consequently been fixed). My guiding principle has been, as Ken Lunde so succinctly put it when I was grumbling one day: "you do the research so your readers don't have to."

Originally, I thought the whole project would take a year at the very most. Boy, was I wrong. Besides the research necessitated by my own ignorance, a few months were lost as priorities shifted after the Kobe earthquake. Also, there's something to be said for experience. I wrote, and threw out, two versions of this book before feeling that I had something worthy to publish. As I found out, there's a big difference between publishing a book and firing off a posting to Usenet. It's been almost two and a half years.

Shoulders to Stand On

As part of my research, both about regular expressions and their history, I have been extremely lucky in the knowledge of others that I have been able to tap. Early on, Tom Wood of Cygnus Systems opened my eyes to the various ways that a regular-expression match engine might be implemented. Vern Paxson (author of *flex)* and Henry Spencer (regular-expression god) have also been a great help. For enlightenment about some of the very early years, before regular expressions entered the realm of computers, I am indebted to Robert Constable and Anil Nerode. For insight into their early computational history, I'd like to thank Brian Kernighan (co-author of *awk*), Ken Thompson (author of *ed* and co-creator of Unix), Michael Lesk (author of *lex),* James Gosling (author of the first Unix version of Emacs, which was also the first to support regular expressions), Richard Stallman (original author of Emacs, and current author of GNU Emacs), Larry Wall (author of *rn, patch,* and Perl), Mark Biggar (Perl's maternal uncle), and Don Libes (author of *Life with Unix,* among others).

The work of many reviewers has helped to insulate you from many of my mistakes. The first line of defense has been my editor, Andy Oram, who has worked tirelessly to keep this project on track and focused. Detailed reviews of the early manuscripts by Jack Halpern saved you from having to see them. In the months the final manuscript was nearing completion, William F. Maton devoted untold hours reviewing *numerous* versions of the chapters. (A detailed review is a lot to ask just once William definitely went above and beyond the call of duty.) Ken Lunde's review turned out to be an incredibly detailed copyedit that smoothed out the English substantially. (Steve Kleinedler did the official copyedit on a later version, from which I learned more about English than I did in 12 years of compulsory education.) Wayne Berke's 25 pages of detailed, insightful comments took weeks to implement, but added substantially to the overall quality. Tom Christiansen's review showed his prestigious skills are not only computational, but linguistic as well: I learned quite a bit about English from him, too. But Tom's skills are computational indeed: discussions resulting from Tom's review were eventually joined in by Larry Wall, who caught a few of my more major Perl gaffes. Mike Stok, Jon Orwant, and Henry Spencer helped with detailed reviews (in particular, I'd like to thank Henry for clarifying some of my misconceptions about the underlying theory). Mike Chachich and Tim O'Reilly also added valuable feedback. A review by experts is one thing, but with a book designed to teach, a review by a non-expert is also important. Jack Halpern helped with the early manuscripts, while Norris Couch and Paul Beard were willing testers of the later manuscript. Their helpful comments allowed me to fill in some of the gaps I'd left.

Errors that might remain

Even with all the work of these reviewers, and despite my best efforts, there are probably still errors to be found in this book. Please realize that none of the reviewers actually saw the very final manuscript, and that there were a few times that I didn't agree with a reviewer's suggestion. Their hard works earns them much credit and thanks, but it's entirely possible that errors were introduced after their review, so any errors that remain are wholly my responsibility. If you do find an error, by all means, please let me know. Appendix A has information on how to contact me.

Appendix A also tells how to get the current errata online. I hope it will be short.

Other Thanks

There are a number of people whose logistic support made this book possible. Ken Lunde of Adobe Systems created custom characters and fonts for a number of the typographical aspects of this book. The Japanese characters are from Adobe Systems' *Heisei Mincho W3* typeface, while the Korean is from the Korean Ministry of Culture and Sports *Munhwa* typeface.

I worked many, *many* hours on the figures for this book. They were nice. Then Chris Reilley stepped in, and in short order whipped some style into them. Almost every figure bears his touch, which is something you'll be grateful for.

I'd like to thank my company, Omron Corporation （オムロン株式会社）, and in particular, 増田清 (Keith Masuda) and 高崎敬雄 (Yukio Takasaki), for their support and encouragement with this project. Having a 900dpi printer at my disposal made development of the special typesetting used in this book possible.

Very special thanks goes to 青山健治 (Kenji Aoyama): the mouse on my ThinkPad broke down as I was preparing the manuscript for final copyedit, and in an unbelievable act of selflessness akin to giving up his firstborn, he loaned me his ThinkPad for the several weeks it took IBM to fix mine. Thanks!

In the Future

I worked on this book for so long, I can't remember what it might feel like to spend a relaxing, lazy day and not feel guilty for it. I plan to enjoy some of the finer luxuries in life (folded laundry, a filled ice tray, a tidy desk), spend a few weekends taking lazy motorcycle rides through the mountains, and take a nice long vacation.

This will be nice, but there's currently a lot of excitement in the regular-expression world, so I won't want to be resting too much. As I go to press, there are active discussions about revamping the regular expression engines with both Python and Perl. My web page (see Appendix A) will have the latest news.

1

Introduction to Regular Expressions

In this chapter:
- ***Solving Real Problems***
- ***Regular Expressions as a Language***
- ***The Regular-Expression Frame of Mind***
- ***Egrep Metacharacters***
- ***Expanding the Foundation***
- ***Personal Glimpses***

Here's the scenario: your boss in the documentation department wants a tool to check for doubled words (such as "this this"), a common problem with documents subject to heavy editing. Your job is to create a solution that will:

- Accept any number of files to check, report each line of each file that has doubled words, highlight (using standard ANSI escape sequences) each doubled word, and ensure that the source filename appears with each line in the report.

- Work across lines, even finding situations when a word at the end of one line is found at the beginning of the next.

- Find doubled words despite capitalization differences, such as with 'The the ', as well as allow differing amounts of *whitespace* (spaces, tabs, newlines, and the like) to lie between the words.

- Find doubled words that might even be separated by HTML tags (and any amount of whitespace, of course). HTML tags are for marking up text on World Wide Web pages, such as to make a word bold: '…`it is <B>very</B> very important`'.

That's certainly a tall order! However, a real problem needs a real solution, and a real problem it is. I used such a tool on the text of this book and was surprised at the way numerous doubled-words had crept in. There are many programming languages one could use to solve the problem, but one with regular expression support can make the job substantially easier.

Regular expressions are the key to powerful, flexible, and efficient text processing. Regular expressions themselves, with a general pattern notation almost like a mini programming language, allow you to describe and parse text. With additional support provided by the particular tool being used, regular expressions can add, remove, isolate, and generally fold, spindle, and mutilate all kinds of text and data. It might be as simple as a text editor's search command or as powerful as a full text processing language. This book will show you the many ways regular expressions can increase your productivity. It will teach you how to *think* regular expressions so that you can master them, taking advantage of the full magnitude of their power.

As we'll see in the next chapter, a full program that solves the doubled-word problem can be implemented in just a few lines of Perl or Python (among others), scripting languages with regular-expression support. With a single regular-expression search-and-replace command, you can find and highlight doubled words in the document. With another, you can remove all lines without doubled words (leaving only the lines of interest left to report). Finally, with a third, you can ensure that each line to be displayed begins with the name of the file the line came from.

The host language (Perl, Python, or whatever) provides the peripheral processing support, but the real power comes from regular expressions. In harnessing this power for your own needs, you will learn how to write regular expressions which will identify text you want, while bypassing text you don't. You'll then combine your expressions with the language's support constructs to actually do something with the text (add appropriate highlighting codes, remove the text, change the text, and so on).

Solving Real Problems

Knowing how to wield regular expressions unleashes processing powers you might not even know were available. Numerous times in any given day, regular expressions help me solve problems both large and small (and quite often, ones that are small but would be large if not for regular expressions). With specific examples that provide the key to solving a large problem, the benefit of regular expressions is obvious. Perhaps not so obvious is the way they can be used throughout the day to solve rather "uninteresting" problems. "Uninteresting" in the sense that such problems are not often the subject of barroom war stories, but quite interesting in that until they're solved, you can't get on with your real work. I find the ability to quickly save an hour of frustration to be somehow exciting.

As a simple example, I needed to check a slew of files (the 70 or so files comprising the source for this book, actually) to confirm that each file contained '`SetSize'` exactly as often (or as rarely) as it contained `'ResetSize'`. To complicate matters, I needed to disregard capitalization (such that, for example, `'setSIZE'` would be counted just the same as `'SetSize'`). The thought of inspecting the 32,000 lines of text by hand makes me shudder. Even using the normal "find this word" search in an editor would have been truly arduous, what with all the files and all the possible capitalization differences.

Regular expressions to the rescue! Typing just a *single,* short command, I was able to check all files and confirm what I needed to know. Total elapsed time: perhaps 15 seconds to type the command, and another 2 seconds for the actual check of all the data. Wow! (If you're interested to see what I actually used, peek ahead to page 32).

As another example, the other day I was helping a friend, Jack, with some email problems on a remote machine, and he wanted me to send a listing of messages in his mailbox file. I could have loaded a copy of the whole file into a text editor and manually removed all but the few header lines from each message, leaving a sort of table of contents. Even if the file wasn't as huge as it was, and even if I wasn't connected via a slow dial-up line, the task would have been slow and monotonous. Also, I would have been placed in the uncomfortable position of actually seeing the text of his personal mail.

Regular expressions to the rescue again! I gave a simple command (using the common search tool *egrep* described later in this chapter) to display the `From:` and `Subject:` line from each message. To tell *egrep* exactly which kinds of lines I did (and didn't) want to see, I used the regular expression ⌈`^(From|Subject):•`⌋ Once Jack got his list, he asked me to send a particular (5,000-line!) message. Again, using a text editor or the mail system itself to extract just the one message would have taken a long time. Rather, I used another tool (one called *sed*) and again used regular expressions to describe exactly the text in the file I wanted. This way, I could extract and send the desired message quickly and easily.

Saving both of us a lot of time and aggravation by using the regular expression was not "exciting," but surely much more exciting than wasting an hour in the text editor. Had I not known regular expressions, I would have never considered that there was an alternative. So, to a fair extent, this story is representative of how regular expressions and associated tools can empower you to do things you might have never thought you wanted to do. Once you learn regular expressions, you wonder how you could ever have gotten by without them.

A full command of regular expressions represents an invaluable skill. This book provides the information needed to acquire that skill, and it is my hope that it will provide the motivation to do so, as well.

Regular Expressions as a Language

Unless you've had some experience with regular expressions, you won't understand the regular expression ⌈`^(From|Subject):•`⌋ from the last example, but there's nothing magic about it. For that matter, there is nothing magic about magic. The magician merely understands something simple which doesn't *appear* to be simple or natural to the untrained audience. Once you learn how to hold a card while making your hand look empty, you only need practice before you, too, can "do magic." Like a foreign language—once you learn it, it stops sounding like gibberish.

The Filename Analogy

Since you have decided to use this book, you probably have at least some idea of just what a "regular expression" is. Even if you don't, you are almost certainly already familiar with the basic concept.

You know that *report.txt* is a specific filename, but if you have had any experience with Unix or DOS/Windows, you also know that the pattern `"*.txt"` can be used to select multiple files. With such filename patterns like this (called *file globs*), there are a few characters* that have special meanings. The star means "match anything," and a question mark means "match any one character." With `"*.txt"`, we start with a match-anything ⌈`*`⌋ and end with the literal ⌈`.txt`⌋, so we end up with a pattern that means "select the files whose names start with anything and end with `.txt"`.

Most systems provide a few additional special characters, but, in general, these filename patterns are limited in expressive power. This is not much of a shortcoming because the scope of the problem (to provide convenient ways to specify groups of files) is limited, well, simply to filenames.

On the other hand, dealing with general text is a much larger problem. Prose and poetry, program listings, reports, lyrics, HTML, articles, code tables, the source to books (such as this one), word lists . . . you name it, if a particular need is specific enough, such as "selecting files," you can develop a specialized scheme or tool. However, over the years, a generalized pattern language has developed which is powerful and expressive for a wide variety of uses. Each program implements and uses them differently, but in general, this powerful pattern language and the patterns themselves are called regular expressions.

---

* The term "character" is pretty loaded in computing, but here I use it merely as a more conversational form of "byte." See "Regular Expression Nomenclature" later in this chapter for details.

The Language Analogy

Full regular expressions are composed of two types of characters. The special characters (like the `*` from the filename analogy) are called ***metacharacters,*** while everything else are called *literal,* or normal text characters. What sets regular expressions apart from filename patterns is the scope of power their metacharacters provide. Filename patterns provide limited metacharacters for limited needs, but a regular expression "language" provides rich and expressive metacharacters for advanced uses.

It might help to consider regular expressions as their own language, with literal text acting as the words and metacharacters as the grammar. The words are combined with grammar according to a set of rules to create an expression which communicates an idea. In the email example, the expression I used to find lines beginning with `'From:•'` or `'Subject:•'` was ⌜`^(From|Subject):•`⌟. The metacharacters are underlined, and we'll get to their interpretation soon.

As with learning any other language, regular expressions might seem intimidating at first. This is why it seems like magic to those with only a superficial understanding, and perhaps completely unapproachable to those that have never seen it at all. But just as 正規表現は簡単だよ!* would soon become clear to a student of Japanese, the regular expression in

```
s!<emphasis>([0-9]+(\.[0-9]+){3})</emphasis>!<inet>$1</inet>!
```

will soon become crystal clear to you, too.

This example is from a Perl language script that my editor used to modify a manuscript. The author had mistakenly used the typesetting tag `<emphasis>` to mark Internet IP addresses (which are sets of periods and numbers that look like `198.112.208.25`). The incantation uses Perl's text-substitution command with the regular expression

€€€€€€€ ⌜`<emphasis>([0-9]+(\.[0-9]+){3})</emphasis>`⌟

to replace such tags with the appropriate `<inet>` tag, while leaving other uses of `<emphasis>` alone. In later chapters, you'll learn all the details of exactly how this type of incantation is constructed, so you'll be able to apply the techniques to your own needs.

---

* "Regular expressions are easy!" A somewhat humorous comment about this: As Chapter 3 explains. the term *regular expression* originally comes from formal algebra. When people ask me what my book is about, the answer "regular expressions" always draws a blank face if they are not already familiar with its use in computers. The Japanese word for regular expression, 正規表現, means as little to the average Japanese as its English counterpart, but my reply in Japanese usually draws a bit more than a blank stare. You see, the "regular" part is unfortunately pronounced identically to a much more common word, a medical term for reproductive organs. You can only imagine what flashes through their minds until I explain!

The goal of this book

The chances that *you* will ever want to replace `<emphasis>` tags with `<inet>` tags is small, but it is very likely that you will run into similar "replace *this* with *that"* problems. The goal of this book is not to teach solutions to specific problems, but rather to teach you how to *think* regular expressions so that you will be able to conquer whatever problem you may face.

The Regular-Expression Frame of Mind

As we'll soon see, complete regular expressions are built up from small building-block units. Each building block is in itself quite simple, but since they can be combined in an infinite number of ways, knowing how to combine them to achieve a particular goal takes some experience.

Don't get me wrong—regular expressions are not difficult to learn and use. In fact, by the end of this very chapter, you'll be able to wield them in powerful ways, even if this is your first real exposure to them.

Still, as with anything, experience helps. With regular expressions, experience can provide a frame of mind, giving direction to one's thinking. This is hard to describe, but it's easy to show through examples. So, in this chapter I would like to quickly introduce some regular-expression concepts. The overview doesn't go into much depth, but provides a basis for the rest of this book to build on, and sets the stage for important side issues that are best discussed before we delve too deeply into the regular expressions themselves.

While some examples may seem silly (because some *are* silly), they really do represent the kind of tasks that you will want to do—you just might not realize it yet. If each and every point doesn't seem to make sense, don't worry too much. Just let the gist of the lessons sink in. That's the goal of this chapter.

If you have some regular-expression experience

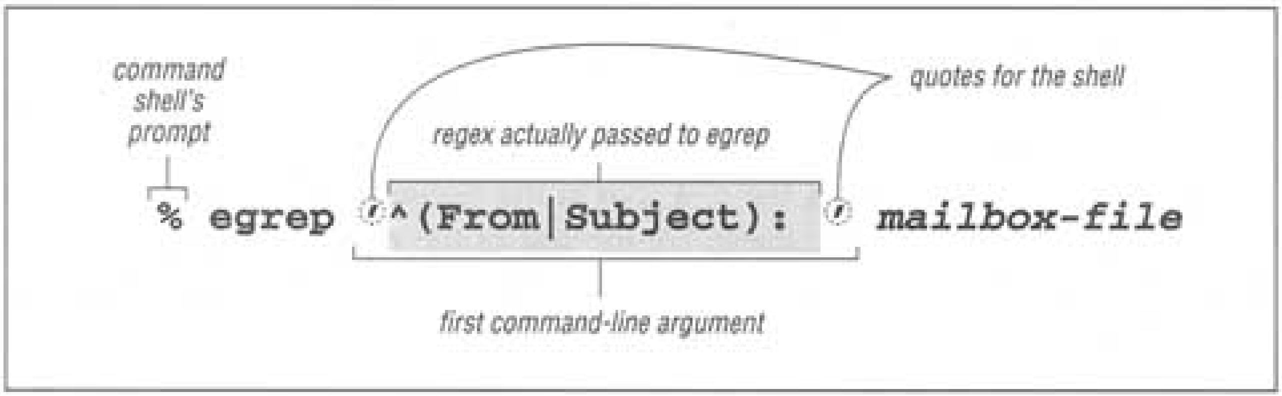
If you're already familiar with regular expressions, much of the overview will not be new, but please be sure to at least glance over it anyway. Although you may be aware of the basic meaning of certain metacharacters, perhaps some of the ways of thinking about and looking at regular expressions will be new.

Just as there is a difference between playing a musical piece well and *making music,* there is a difference between understanding regular expressions and *really understanding* them. Some of the lessons present the same information that you are already familiar with, but in ways that may be new and which are the first steps to *really understanding.*

Searching Text Files: Egrep

Finding text is one of the simplest uses of regular expressions—many text editors and word processors allow you to search a document using a regular-expression pattern. Even more simple is the utility *egrep.** Give *egrep* a regular expression and some files to search, and it attempts to match the regular expression to each line of each file, displaying only those lines that are successfully matched.

Returning to the initial email example, the command I actually used is shown in Figure 1-1. *egrep* interprets the first command-line argument as a regular expression, and any remaining arguments as the file(s) to search. Note, however, that the quotes shown in Figure 1-1 are *not* part of the regular expression, but are needed by my command shell.** When using *egrep*, I almost always wrap the regular expression with quotes like this.



```
% egrep '^(From|Subject): ' mailbox-file
```

Figure 1-1:
Invoking egrep from the command line

If your regular expression doesn't use any of the dozen or so metacharacters that *egrep* understands, it effectively becomes a simple "plain text" search. For example, searching for ⌈`cat`⌋ in a file finds and displays all lines with the three letters `c·a·t` in a row. This includes, for example, any line containing `vacation`.

Even though the line might not have the *word* `cat`, the `c·a·t` sequence in `vacation` is still enough to be matched. That's the only thing asked for, and since it's there, *egrep* displays the whole line. The key point is that regular-expression searching is not done on a "word" basis—*egrep* can understand the concept of

* *egrep* is freely-available for many systems, including DOS, MacOS, Windows, Unix, and so on (see Appendix A for information on how to obtain a copy of *egrep* for your system). Some users might be more familiar with the program e*grep*, which is similar in many respects. The discussion of the regular-expression landscape in Chapter 3 makes it clear why I chose *egrep* to begin with.

** The command shell is that part of the system that accepts your typed commands and actually executes the programs you request. With the shell I use, the quotes serve to group the command argument, telling the shell not to pay too much attention to what's inside (such as, for example, not treating `*.txt` as a filename pattern so that it is left for *egrep* to interpret as it sees fit, which for *egrep* means a regular expression). DOS users of COMMAND.COM should probably use doublequotes instead.

bytes and lines in a file, but it generally has no idea of English (or any other language's) words, sentences, paragraphs, or other high-level concepts.*

Egrep Metacharacters

Let's start to explore some of the *egrep* metacharacters that supply its regular-expression power. There are various kinds which play various roles. I'll go over them quickly with a few examples, leaving the detailed examples and descriptions for later chapters.

Before we begin, please make sure to review the typographical conventions explained in the preface (on page *xx*). This book forges a bit of new ground in the area of typesetting, so some of my notations will be unfamiliar at first.

Start and End of the Line

Probably the easiest metacharacters to understand are ⌈`^`⌋ (*caret*) and ⌈`$`⌋ (dollar), which represent the start and end, respectively, of the line of text as it is being checked. As we've seen, the regular expression ⌈`cat`⌋ finds `c·a·t` anywhere on the line, but ⌈`^cat`⌋ matches only if the `c·a·t` is at the beginning of the line—the ⌈`^`⌋ is used to effectively anchor the match (of the rest of the regular expression) to the start of the line. Similarly, ⌈`cat$`⌋ finds `c·a·t` only at the end of the line, such as a line ending with `scat`.

Get into the habit of interpreting regular expressions in a rather literal way. For example, don't think

> ⌈`^cat`⌋ matches a line with cat at the beginning

but rather:

> ⌈`^cat`⌋ matches if you have the beginning of a line, followed immediately by `c`,
> followed immediately by `a`, followed immediately by `t`.

They both end up meaning the same thing, but reading it the more literal way allows you to intrinsically understand a new expression when you see it. How would you read ⌈`^cat$`⌋, ⌈`^$`⌋, or even simply ⌈`^`⌋ alone? ❖ Turn the page to check your interpretations.

The caret and dollar are particular in that they match a *position* in the line rather than any actual text characters themselves. There are, of course, various ways to actually match real text. Besides providing literal characters in your regular expression, you can also use some of the items discussed in the next few sections.

---

* The idea is that *egrep* breaks the input file into separate text lines and then checks them with the regular expression. Neither of these phases attempts to understand the human units of text such as sentences and words. I was struggling for the right way to express this until I saw the phrase "high-level concepts" in Dale Dougherty's *sed & auk* and felt it fit perfectly.

Character Classes

Matching any one of several characters

Let's say you want to search for "grey," but also want to find it if it were spelled "gray". The ⌈`[…]`⌋ construct, usually called a *character class,* lets you list the characters you want to allow at that point: ⌈`gr[ea]y`⌋. This means to find "`g`, followed by `r`, followed by an `e` or an `a`, all followed by `y`." I am a really poor speller, so I'm always using regular expressions like this to check a word list for proper spellings. One I use often is ⌈`sep[ea]r[ea]te`⌋, because I can never remember whether the word is spelled "seperate," "separate," "separete,' or what.

As another example, maybe you want to allow capitalization of a word's first letter: ⌈`[Ss]mith`⌋. Remember that this still matches lines that contain `smith` (or `Smith`) embedded within another word, such as with `blacksmith`. I don't want to harp on this throughout the overview, but this issue does seem to be the source of problems among some new users. I'll touch on some ways to handle this embedded-word problem after we examine a few more metacharacters.

You can list in the class as many characters as you like. For example, ⌈`[123456]`⌋ matches any of the listed digits. This particular class might be useful as part of ⌈`<H[123456]>`⌋, which matches `<H1>`, `<H2>`, `<H3>`, etc. This can be useful when searching for HTML headers.

Within a character class, the *character-class metacharacter* '-' (*dash*) indicates a range of characters: ⌈`<H[1-6]>`⌋ is identical to the previous example. ⌈`[0-9]`⌋ and ⌈`[a-z]`⌋ are common shorthands for classes to match digits and lowercase letters, respectively. Multiple ranges are fine, so ⌈`[0123456789abcdefABCDEF]`⌋ can be written as ⌈`[0-9a-fA-F]`⌋. Either of these can be useful when processing hexadecimal numbers. You can even combine ranges with literal characters: ⌈`[0-9A-Z_!.?]`⌋ matches a digit, uppercase letter, underscore, exclamation point, period, or a question mark.

Note that a dash is a metacharacter only within a character class—otherwise it matches the normal dash character. In fact, it is not even always a metacharacter within a character class. If it is the first character listed in the class, it can't possibly indicate a range, so it is not considered a metacharacter.

> *Consider character classes as their own mini language. The rules regarding which metacharacters are supported (and what they do) are completely different inside and outside of character classes.*

We'll see more examples of this shortly.

Reading ⌈`^cat$`⌋, ⌈`^$`⌋, and ⌈`^`⌋

❖ Answers to the questions on page 8.

⌈`^cat$`⌋

Literally: matches if the line has a beginning-of-line (which, of course, all lines have), followed immediately by `c·a·t` , and then followed immediately by the end of the line.

Effectively means: a line that consists of only `cat`—no extra words, spaces, punctuation . . . nothing.

⌈`^$`⌋

Literally: matches if the line has a beginning-of-line, followed immediately by the end of the line.

Effectively means: an empty line (with nothing in it, not even spaces).

⌈`^`⌋

Literally: matches if the line has a beginning-of-line.

Effectively meaningless! Since every line has a beginning, every line will match—even lines that are empty!

Negated character classes

If you use ⌈`[^…]`⌋ instead of ⌈`[…]`⌋, the class matches any character that *isn't* listed. For example, ⌈`[^1-6]`⌋ matches a character that's *not* 1 through 6. More or less, the leading ^ in the class "negates" the list—rather than listing the characters you want to include in the class, you list the characters you don't want to be included.

You might have noticed that the ^ used here is the same as the start-of-line caret. The character is the same, but the meaning is completely different. Just as the English word "wind" can mean different things depending on the context (sometimes a strong breeze, sometimes what you do to a clock), so can a metacharacter. We've already seen one example, the range-building dash. It is valid only inside a character class (and at that, only when not first inside the class). ^ is an anchor outside a class, a class metacharacter inside, but only when it is immediately after the class's opening bracket (otherwise, inside a class it's not special).

Returning to our word list, we know that in English, the letter `q` is almost always followed by u. Let's search for odd words that have `q` followed by something else—translating into a regular expression, that becomes ⌈`q[^u]`⌋. I tried it on my word list, and there certainly aren't many! I did find a few, and this included a number of words that I didn't even know were English.

Here's what I typed:

```
% egrep 'q[^u]' word.list
Iraqi
Iraqian
miqra
qasida
qintar
qoph
zaqqum
%
```

Two notable words not listed are "Qantas", the Australian airline, and "Iraq".

Although both are in my list, neither were listed. Why? ❖ Think about it for a bit, then turn the page to check your reasoning.

Remember that a negated character class means "match a character that's not listed" and not "don't match what is listed." These might seem the same, but the `Iraq` example shows the subtle difference. A convenient way to view a negated class is that it is simply a shorthand for a normal class which includes all possible characters *except* those that are listed.

Matching Any Character—Dot

The metacharacter⌜`.`⌟ (usually called *dot)* is a shorthand for a character class that matches any character. It can be convenient when you want to have an "any character here" place-holder in your expression. For example, if you want to search for a date such as `07/04/76, 07-04-76`, or even `07.04.76`, you could go to the trouble to construct a regular expression that uses character classes to explicitly allow `'/'`, `'-'`, or `'.'` between each number, such as ⌜`07[-./]04[-./]76`⌟. You could also try simply using ⌜`07.04.76`⌟.

Quite a few things are going on with this example that might be unclear at first. `In` ⌜`07 [-./] 04 [-./] 76`⌟, the dots are *not* metacharacters because they are within a character class. (Remember, the list of metacharacters and their meanings are different inside and outside of character classes.) The dashes are also not metacharacters, although within a character class they normally are. As I've mentioned, a dash is not special when it is the first character in the class.

With ⌈`07.04.76`⌋, the dots *are* metacharacters that match any character, including the dash, period, and slash that we are expecting. However, it is important to know that each dot can match any character at all, so it can match, say, 'lottery numbers: `19 207304 7639'`.

⌈`07[-./]04[-./]76`⌋ is more precise, but it's more difficult to read and write. ⌈`07.04.76`⌋ is easy to understand, but vague. Which should we use? It all depends upon what you know about the data you are searching, and just how specific you feel you need to be. One important, recurring issue has to do with balancing your

Why doesn't ⌈`q[^u]`⌋ match 'Qantas' or 'Iraq?

❖ Answer to the question on page 11.

Qantas didn't match because the regular expression called for a lowercase `q`, whereas the `Q` in Qantas is uppercase. Had we used ⌈`Q[^u]`⌋ instead, we would have found it, but not the others, since they don't have an uppercase `Q`. The expression ⌈`[Qq][^u`⌋ would have found them all.

The `Iraq` example is somewhat of a trick question. The regular expression calls for `q` followed by a character that's not `u`. But because egrep strips the end-of-line character(s) before checking with the regular expression (a little fact I neglected to mention, sorry!) there's no data at all after the `q`. Yes, there is no `u` after the `q`, but there's no non-`u` either!

Don't feel too bad because of the trick question.* Let me assure you that had egrep not stripped the newlines (as some other tools don't), or had `Iraq` been followed by spaces or other words or whatnot, the line would have matched. It is eventually important to understand the little details of each tool, but at this point what I'd like you to come away with from this exercise is that a character class, even negated, still requires a character to match.

knowledge of the text being searched against the need to always be exact when writing an expression. For example, if you know that with your data it would be highly unlikely for ⌈`07.04.76`⌋ to match in an unwanted place, it would certainly be reasonable to use it. Knowing the target text well is an important part of wielding regular expressions effectively.

Alternation

Matching any one of several subexpressions

A very convenient metacharacter is ⌈`|`⌋, which means "or" It allows you to combine multiple expressions into a single expression which matches any of the individual expressions used to make it up. For example, ⌈`Bob`⌋ and ⌈`Robert`⌋ are two separate expressions, while ⌈`Bob|Robert`⌋ is one expression that matches either. When combined this way, the subexpressions are called *alternatives.*

Looking back to our ⌈`gr[ea]y`⌋ example, it is interesting to realize that it can be written as ⌈`grey|gray`⌋, and even ⌈`gr(a|e)y`⌋. The latter case uses parentheses to constrain the alternation. (For the record, parentheses are metacharacters too.)

---

*Once, in fourth grade, I was leading the spelling bee when I was asked to spell "miss". " `m·i·s·s` was my answer. Miss Smith relished in telling me that no, it was " `M·i·s·s` " with a capital M, that I should have asked for an example sentence, and that I was out. It was a traumatic moment in a young boy's life. After that, I never liked Miss Smith, and have since been a very poor speler.

Without the parentheses, ⌈gra|ey⌋ means "⌈gra⌋ or ⌈ey⌋", which is not what we want here. Alternation reaches far, but not beyond parentheses. Another example is ⌈(First|1st)•[Ss]treet⌋. Actually, since both ⌈First⌋ and ⌈1st⌋ end with ⌈st⌋, they can be shortened to ⌈(Fir |1)st•[Ss]treet⌋, but that's not necessarily quite as easy to read. Still, be sure to understand that they do mean the same thing.

Although the ⌈gr[ea]y⌋ versus ⌈gr(a|e)y⌋ examples might blur the distinction, be careful not to confuse the concept of alternation with that of a character class. A character class represents a single character in the target text. With alternation, each alternative can be a full-fledged regular expression in and of itself. Character classes are almost like their own special mini-language (with their own ideas about metacharacters, for example), while alternation is part of the "main" regular expression language. You'll find both to be extremely useful.

Also, take care when using caret or dollar in an expression with alternation. Compare ⌈^From|Subject|Date:•⌋ with ⌈^(From|Subject|Date):•⌋. Both appear similar to our earlier email example, but what each matches (and therefore how useful it is) differs greatly. The first is composed of three plain alternatives, so it will match when we have "⌈^From⌋ or ⌈Subject⌋ or ⌈Date:•⌋, " which is not particularly useful. We want the leading caret and trailing ⌈:•⌋ to apply to each alternative. We can accomplish this by using parentheses to "constrain" the alternation:

⌈^(From|Subject |Date):•⌋

This matches:

1) start-of-line, followed by F·r·o·m, followed by ':•'

or *2)* start-of-line, followed by S·u·b·j·e·c·t, followed by ':•'

or *3)* start-of-line, followed by D·a·t·e , followed by ':•'

As you can see, the alternation happens within the parentheses, which effectively allows the wrapping ⌈`^…:•`⌋ to apply to each alternative, so we can say it matches:

€€€ ⌈`^(From):•`⌋ or ⌈`^(Subject):•`⌋ or ⌈`^(Date):•`⌋

Putting it less literally: it matches any line that begins with '`From:•`', '`Subject:•`', or '`Date:•`', which is exactly what would be useful to get a listing of the messages in an email file.

Here's an example:

```
% egrep  '^(From|SubjectlDate): '  mailbox
From: elvis@tabloid.org (The King)
Subject: be seein' ya around
Date: Thu, 31 Oct 96 11:04:13
From: The Prez <president@whitehouse.gov>
Date: Tue, 5 Nov 1996 8:36:24
Subject: now, about your vote...
   ⋮
```

Word Boundaries

A common problem is that a regular expression that matches the word you want can often also match where the "word" is embedded within a larger word. I mentioned this in the `cat`, `gray`, and `Smith` examples, but despite what I said about *egrep* generally not having any concept of words, it turns out that some versions of *egrep* offer limited support: namely the ability to match the boundary of a word (where a word begins or ends).

You can use the (perhaps odd looking) *metasequences* 「`\<`」 and 「`\>`」 if your version happens to support them (not all versions of *egrep* do). Word-based equivalents of 「`^`」 and 「`$`」, they match the position at the start and end of a word, respectively (so, like the line anchors caret and dollar, they never actually consume any characters). The expression 「`\<cat\>`」 literally means "match if we can find a start-of-word position, followed immediately by `c·a·t` , followed immediately by an end-of-word position ." More naturally, it means "find the word cat." If you wanted, you could use 「`\<cat`」 or 「`cat\>`」 to find words starting and ending with cat.

Note that 「`<`」 and 「`>`」 alone are not metacharacters when combined with a backslash, the *sequences* become special. This is why I called them "metasequences." It's their special interpretation that's important, not the number of characters, so for the most part I use these two meta-words interchangeably.

Remember, not all versions of *egrep* support these word-boundary metacharacters, and those that do don't magically understand the English language. The "start of a word" is simply the position where a sequence of alphanumeric characters begins; "end of word" being where such a sequence ends. Figure 1-2 shows a sample line with these positions marked.

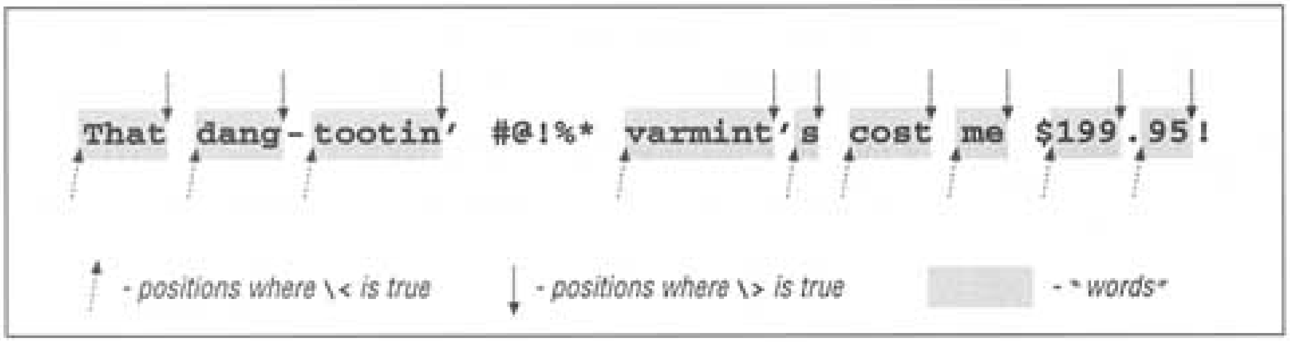

That dang-tootin' #@!%* varmint's cost me $199.95!
- positions where \< is true
- positions where \> is true
- "words"

Figure 1-2:
Start and end of "word" positions

The word-starts (as *egrep* recognizes them) are marked with up arrows, the word-ends with down arrows. As you can see, "start and end of word" is better phrased as "start and end of an alphanumeric sequence," but perhaps that's too much of a mouthful.

In a Nutshell

Table 1-1: Summary of Metacharacters Seen So Far

Metacharacter	**Name**	**Meaning**
`.`	*dot*	any on character
`[…]`	*character class*	any character listed
`[^…]`	*negated character class*	any character not listed
`^`	*caret*	the position at the start of the line
`$`	*dollar*	the position at the end of the line
`\<`	*backslash less-than*	*the position at the start of a word
`\>`	*backslash greater-than*	*the position at the end of a word
		**not supported by all versions of egrep*
`|`	*or; bar*	matches either expression it separates
`(…)`	*parentheses*	used to limit the scope of ⌈|⌋,plus additional uses yet to be discussed

Table 1-1 summarizes the metacharacters we have seen so far. In addition to the table, some important points to remember are:

- The rules about which characters are and aren't metacharacters (and exactly what they mean) are different inside a character class. For example, dot is a metacharacter outside of a class, but not within one. Conversely, a dash is a metacharacter within a class, but not outside. And a caret has one meaning outside, another if specified inside a class immediately after the opening [, and a third if given elsewhere in the class.

- Don't confuse alternation with a character class. The class ⌈`[abc]`⌋ and the alternation ⌈`(a|b|c)`⌋ effectively mean the same thing, but the similarity in this example does not extend to the general case. A character class can match exactly one character, and that's true no matter how long or short the specified list of acceptable characters might be. Alternation, on the other hand, can have arbitrarily long alternatives, each textually unrelated to the other: ⌈`\<(1,000,000|million|thousand•thousand)\>`⌋. But alternation can't be negated like a character class.

- A negated character class is simply a notational convenience for a normal character class that matches everything not listed. Thus, ⌈`[^x]`⌋ doesn't mean " match unless there is an `x` ," but rather "match if there is something that is not `x`." The difference is subtle, but important. The first concept matches a blank line, for example, while, in reality, ⌈`[^x]`⌋ does not.

What we have seen so far can be quite useful, but the real power comes from *optional* and *counting* elements.

Optional Items

Let's look at matching `color` or `colour`. Since they are the same except that one has a u and the other doesn't, we can use ⌈`colou?r`⌋ to match either. The metacharacter ⌈`?`⌋ (*question mark*) means *optional*. It is placed after the character that is allowed to appear at that point in the expression, but whose existence isn't actually required to still be considered a successful match.

Unlike other metacharacters we have seen so far, the question-mark attaches only to the immediately-preceding item. Thus ⌈`colou?r`⌋ is interpreted as " ⌈`c`⌋, then ⌈`o`⌋ then ⌈`l`⌋ then ⌈`o`⌋ then ⌈`u?`⌋ then ⌈`r`⌋. "

The ⌈`u?`⌋ part will always be successful: sometimes it will match a u in the text, while other times it won't. The whole point of the ?-optional part is that it's successful either way. This isn't to say that any regular expression that contains ? will always be successful. For example, against '`semicolon`', both ⌈`colo`⌋ and ⌈`u?`⌋ are successful (matching `colo` and nothing, respectively). However, the final ⌈`r`⌋ will fail, and that's what disallows semicolon, in the end, from being matched by ⌈`colou?r`⌋.

As another example, consider matching a date that represents July fourth, with the July part being either `July` or `Jul`, and the fourth part being `fourth`, `4th`, or simply `4`. Of course, we could just use ⌈`(July|Jul)•(fourth|4th|4)`⌋, but let's explore other ways to express the same thing.

First, we can shorten the ⌈`(July\Jul)`⌋ to ⌈`(July?)`⌋. Do you see how they are effectively the same? The removal of the ⌈`|`⌋ means that the parentheses are no longer really needed. Leaving the parentheses doesn't hurt, but ⌈`July?`⌋, with them removed, is a bit less cluttered. This leaves us with ⌈`July?•(fourth|4th|4)`⌋.

Moving now to the second half, we can simplify the ⌈`4th|4`⌋ to ⌈`4(th)?`⌋. As you can see, ⌈`?`⌋ can attach to a parenthesized expression. Inside the parentheses can be as complex a subexpression as you like, but "from the outside" it is considered a unit. Grouping for ⌈`?`⌋ (and other similar metacharacters that I'll introduce momentarily) is one of the main uses of parentheses.

Our expression now looks like ⌈`July?·(fourth|4(th)?)`⌋. Although there are a fair number of metacharacters, and even nested parentheses, it is not that difficult to decipher and understand. This discussion of two essentially simple examples has been rather long, but in the meantime we have covered tangential topics that add a lot, if perhaps only subconsciously, to our understanding of regular expressions. It's easier to start good habits early, rather than to have to break bad ones later.

Other Quantifiers: Repetition

Similar to the question mark are ⌈`+`⌋ (*plus*) and ⌈`*`⌋ (an asterisk, but as a regular-expression metacharacter, I prefer the term *star*). The metacharacter ⌈`+`⌋, means "one or more of the immediately-preceding item," and ⌈`*`⌋ means "any number, including none, of the item." Phrased differently, ⌈`…*`⌋ means "try to match it as many times as possible, but it's okay to settle for nothing if need be." The construct with plus, ⌈`+`⌋, is similar in that it will also try to match as many times as possible, but different in that it will fail if it can't match at least once. These three metacharacters, question mark, plus, and star, are called *quantifiers* (because they influence the quantity of a match they govern).

Like ⌈`…?`⌋, the ⌈`…*`⌋ part of a regular expression will always succeed, with the only issue being what text (if any) will be matched. Contrast this to ⌈`…+`⌋ which fails unless the item matches at least once.

An easily understood example of star is ⌈`•*`⌋, the combination with a space allowing optional spaces. (⌈`•?`⌋ allows one optional space, while ⌈`•*`⌋ allows any number.) We can use this to make page 9's `<H[1-6]>` example flexible. The HTML specification* says that spaces are allowed immediately before the closing >, such as with `<H3•>` and `<H4•••>`. Inserting ⌈`•*`⌋ into our regular expression where we want to allow (but not require) spaces, we get ⌈`<H[1-6]•*>`⌋. This still matches `<H1>`, as no spaces are required, but it also flexibly picks up the other versions.

Exploring further, let's search for a particular HTML tag recognized by Netscape's World Wide Web browser *Navigator*. A tag such as `<HR•SIZE=14>` indicates that a line (a Horizontal Rule) 14 pixels thick should be drawn across the screen. Like the `<H3>` example, optional spaces are allowed before the closing angle bracket. Additionally, they are allowed on either side of the equal sign. Finally, one space is required between the `HR` and `SIZE`, although more are allowed. For this last case, we could just insert ⌈`••*`⌋, but instead let's use ⌈`•+`⌋. The plus allows extra spaces while still requiring at least one. Do you see how this will match the same as ⌈`••*`⌋? This leaves us with ⌈`<HR•+SIZE•*=•*14•*>`⌋.

Although flexible with respect to spaces, our expression is still inflexible with respect to the size given in the tag. Rather than find tags with only a particular size (such as `14`), we want to find them all. To accomplish this, we replace the ⌈`14`⌋ with an expression to find a general number. Well, a number is one or more digits. A digit is ⌈`[0-9]`⌋, and "one or more" adds a plus, so we end up replacing ⌈`14`⌋ by ⌈`[0-9]+`⌋. As you can see, a single character class is one "unit", so can be subject directly to plus, question mark, and so on, without the need for parentheses.

---

* If you are not familiar with HTML, never fear. I use these as real-world examples, but I provide all the details needed to understand the points being made. Those familiar with parsing HTML tags will likely recognize important considerations I don't address at this point in the book.

This leaves us with ⌈`<HR•+SIZE•*=•*[0-9]+•*>`⌋, which is certainly a mouthful. It looks particularly odd because the subject of most of the stars and pluses are space characters, and our eye has always been trained to treat spaces specially. That's a habit you will have to break when reading regular expressions, because the space character is a normal character, no different from, say, `j` or `4`.

Continuing to exploit a good example, let's make one more change. With Navigator, not only can you use this sized-`HR` tag, but you can still use the standard sizeless version that looks simply like `<HR>` (with extra spaces allowed before the >, as always). How can we modify our regular expression so that it matches either type? The key is realizing that the size part is *optional* (that's a hint). ❖ Turn the page to check your answer.

Take a good look at our latest expression (in the answer box) to appreciate the differences among the question mark, star, and plus, and what they really mean in practice. Table 1-2 summarizes their meanings. Note that each quantifier has some minimum number of matches that it needs to be considered a successful match, and a maximum number of matches that it will ever attempt. With some, the minimum number is zero; with some, the maximum number is unlimited.

Table 1-2: Summary of Quantifier "Repetition Metacharacters"

	Minimum Required	**Maximum to Try**	**Meaning**
`?`	none	1	one allowed; none required (*"one optional"*)
`*`	none	no limit	unlimited allowed; none required (*"any amount optional"*)
`+`	1	no limit	one required; more allowed (*"some required"*)

Defined range of matches: intervals

Some versions of egrep support a metasequence for providing your own minimum and maximum : ⌈¼`{`*`min,max`*`}`⌋. This is called the *interval* quantifier. For example, ⌈¼`{3,12}`⌋ matches up to 12 times if possible, but settles for three. Using this notation, `{0,1}` is the same as a question mark.

Not many versions of *egrep* support this notation yet, but many other tools do, so I'll definitely discuss it in Chapter 3 when I look in detail at the broad spectrum of metacharacters in common use today.

Ignoring Differences in Capitalization

HTML tags can be written with mixed capitalization, so `<h3>` and `<Hr•Size=26>` are both legal. Modifying `<H[1-6]•*>` would be a simple matter of using ⌈`[Hh]`⌋ for ⌈`H`⌋, but it becomes more troublesome with the longer ⌈`HR`⌋ and ⌈`SIZE`⌋ of the other example. Of course, we could use ⌈`[Hh][Rr]`⌋ and ⌈`[Ss][Ii][Zz][Ee]`⌋, but it is easier to tell *egrep* to ignore case by performing the match in a *case insensitive* manner in which capitalization differences are simply ignored.

This is not a part of the regular-expression language, but is a related useful feature many tools provide. Use the -i option to tell *egrep* to do a case-insensitive match. Place `-i` on the command line before the regular expression:

```
% egrep -i '<HR( +SIZE *= *[0-9]+)? *>'  file
```

In future chapters, we will look at many convenient support features of this kind.

Parentheses and Backreferences

So far, we have seen two uses for parentheses: to limit the scope of `|`, and to provide grouping for quantifiers, such as question mark and star, to work with. I'd like to discuss another specialized use that's not common in *egrep* (although GNU's popular version does support it), but which is commonly found in many other tools.

Parentheses can "remember" text matched by the subexpression they enclose. We'll use this in a partial solution to the doubled-word problem at the beginning of this chapter. If you knew the the specific doubled word to find (such as "the" earlier in this sentence—did you catch it?), you could search for it explicitly, such as with ⌈`\<the•the>\`⌋ In this case, you would also find items such as `the•theory`, but you could easily get around that problem if your *egrep* supports the ⌈`\<...\>`⌋ mentioned earlier: ⌈`\<the•the\>`⌋. We could even use ⌈`•+`⌋ for the space to make it more flexible.

However, having to check for every possible pair of words would be an impossible task. Wouldn't it be nice if we could match one generic word, and then say "now match the same thing again"? If your *egrep* supports *backreferencing*, you can. Backreferencing is a regular-expression feature which allows you to match new text that is the same as some text matched earlier in the expression without specifically knowing the text when you write the expression.

We start with ⌈\<the•+the\>⌋ and replace the initial [the] with a regular expression to match a general word, say ⌈[A-Za-z]+⌋. Then, for reasons that will become clear in the next paragraph, let's put parentheses around it. Finally, we replace the second 'the' by the special metasequence ⌈\1⌋. This yields ⌈\<**(**A-Za-z]+**)**•+**\1**\>⌋.

With tools that support backreferencing, parentheses "remember" the text that the subexpression inside them matches, and the special metasequence ⌈\1⌋ represents that text later in the regular expression, whatever it happens to be at the time.

You can have more than one set of parentheses, of course, and you use ⌈\1⌋, ⌈\2⌋, ⌈\3⌋, etc., to refer to the set that matched the text you want to match with. Pairs of parentheses are numbered by counting opening parentheses from the left.

Making a subexpression optional

❖ Answer to the question on page 18.

In this case, "optional" means that it is allowed once, but is not required. That means using ⌈`?`⌋.

Since the thing that's optional is larger than one character, we must use parentheses: ⌈`(…) ?`⌋. Inserting into our expression, we get:

⌈`<HR(•+SIZE•*=•*[0-9]+)?•*>`⌋

Note that the ending ⌈`•*`⌋ is kept outside of the ⌈`(…)?`⌋. This still allows something such as `<HR•>`. Had we included it within the parentheses, ending spaces would have been allowed only when the size component was present.

With our '`the•the`' example, ⌈`[A-Za-z]+`⌋ matches the first `the`. It is within the first set of parentheses, so the '`the`' matched becomes available via ⌈`\1`⌋ if the following ⌈`•+`⌋ matches, the subsequent ⌈`\1`⌋ will require 'the'. If successful, ⌈`\>`⌋ then makes sure that we are now at an end-of-word boundary (which we wouldn't be were the text `the•theft`). If successful, we've found a repeated word. It's not always the case that that is an error (such as with "that" in this sentence), but once the suspect lines are shown, you can peruse what's there and decide for yourself.

When I decided to include this example, I actually tried it on what I had written so far. (I used a version of *egrep* which supports both ⌈`\<…\>`⌋ and backreferencing.) To make it more useful, so that '`The•the`' would also be found, I used the case-insensitive -i option mentioned earlier:

```
% egrep -i '\<([a-z]+) +\1\>' files...
```

I'm somewhat embarrassed to say that I found fourteen sets of mistakenly 'doubled doubled' words!

As useful as this regular expression is, it is important to understand its limitations. Since *egrep* considers each line in isolation, it isn't able to find when the ending word of one line is repeated at the beginning of the next. For this, a more flexible tool is needed, and we will see some examples in the next chapter.

The Great Escape

One important thing I haven't mentioned yet is how to actually match a character that in a regular expression would normally be interpreted as a metacharacter. For example, if I wanted to search for the Internet hostname `ega.att.com` using ⌈`ega.att.com`⌋, I might end up matching something like `megawatt•computing`. Remember, ⌈`.`⌋ is a metacharacter that matches any character.

The metasequence to match an actual period is a period preceded by a backslash: ⌜`ega\.att\.com`⌟. The ⌜`\.`⌟ is called an *escaped period*, and you can do this with all the normal metacharacters except in a character-class. When a metacharacter is escaped, it loses its special meaning and becomes a literal character. If you like, you can consider the sequence to be a special metasequence to match the literal character. It's all the same.

As another example, you could use ⌜`\([a-zA-Z]+\)`⌟ to match a word within parentheses, such as '`(very)`'. The backslashes in the ⌜`\(`⌟ and ⌜`\)`⌟ sequences remove the special interpretation of the parentheses, leaving them as literals to match parentheses in the text.

When used before a non-metacharacter, a backslash can have different meanings depending upon the version of the program. For example, we have already seen how some versions treat ⌜`\<`⌟, ⌜`\>`⌟, ⌜`\1`⌟, etc. as metasequences. We will see more examples in later chapters.

Expanding the Foundation

I hope the examples and explanations so far have helped to establish the basis for a solid understanding of regular expressions, but please realize that what I've provided so far lacks depth. There's so much more out there.

Linguistic Diversification

I mentioned a number of regular expression features that most versions of *egrep* support. There are other features, some of which are not supported by all versions, that I'll leave for later chapters.

Unfortunately, the regular expression language is no different from any other in that it has various dialects and accents. It seems each new program employing regular expressions devises its own "improvements." The state of the art continually moves forward, but changes over the years have resulted in a wide variety of regular expression "flavors." We'll see many examples in the following chapters.

The Goal of a Regular Expression

From the broadest top-down view, a regular expression either matches a lump of text (with *egrep,* each line) or it doesn't. When crafting a regular expression, you must consider the ongoing tug-of-war between having your expression match the lines you want, yet still not matching lines you don't want.

Also, while *egrep* doesn't care where in the line the match occurs, this concern is important for many other regular-expression uses. If your text is something such as

```
...zip is 44272. If you write, send $4.95 to cover postage
and...
```

and you merely want to find lines matching ⌈`[0-9]+`⌋, you don't care which numbers are matched. However, if your intent is to *do something* with the number (such as save to a file, add, replace, and such—we will see examples of this kind of processing in the next chapter), you'll care very much exactly *which* numbers are matched.

A Few More Examples

As with any language, experience is *a very good thing*, so I'm including a few more examples of regular expressions to match some common constructs.

Half the battle when writing regular expressions is getting successful matches when and where you want them. The other half is to *not* match when and where you don't want. In practice, both are important, but for the moment I would like to concentrate on the "getting successful matches" aspect. Not taking the examples to their fullest depths doesn't detract from the experiences we do take from them.

Variable names

Many programming languages have identifiers (variable names and such) that are allowed to contain only alphanumeric characters and underscores, but which may not begin with a number, that is, ⌈`[a-zA-Z_][a-zA-Z_0-9]*`⌋. The first class matches what the first character can be, the second (with its accompanying star) allows the rest of the identifier. If there is a limit on the length of an identifier, say 32 characters, you might replace the star with ⌈`{0,31}`⌋ if the ⌈`{min,max}`⌋ notation is supported. (This construct, the interval quantifier, was briefly mentioned on page 18.)

A string within doublequotes

A simple solution might be: ⌜`"[^"]*"`⌟

The quotes at either end are to match the open and close quote of the string. Between them, we can have anything. . . except another doublequote! So, we use ⌜`[^"]`⌟ to match all characters except a doublequote, and apply using a star to indicate we can have any number of such non-doublequote characters.

A more useful (but more complex) definition of a doublequoted string allows doublequotes within the string if they are escaped with a backslash, such as in `"nail•the•2\"x4\"•plank"`. We'll see this example again in Chapters 4 and 5 during the discussion of the details about how a match is actually carried out.

Dollar amount (with optional cents)

One approach is: ⌈`\$[0-9]+(\.[0-9][0-9])?`⌋

From a top-level perspective, this is a simple regular expression with three parts: ⌈`\$`⌋ and ⌈`…+`⌋ and ⌈`(…)?`⌋, which might be loosely paraphrased as "A literal dollar sign, a bunch of one thing, and finally perhaps another thing." In this case, the "one thing" is a digit (with a bunch of them being a number), and "another thing" is the combination of a decimal point followed by two digits.

This example is a bit naive for several reasons. For instance, with *egrep,* you care only whether there's a match or not, not how much (nor how little) is actually matched, so bothering to allow optional cents makes no sense. (Ignoring them doesn't change which lines do and don't match.) If, however, you need to find lines that contain just a price, and nothing else, you can wrap the expression with ⌈`^…$`⌋. In such a case, the optional cents part becomes important since it might or might not come between the dollar amount and the end of the line.

One type of value our expression won't match is `'$.49'`. To solve this, you might be tempted to change the plus to a star, but that won't work. As to why, I'll leave it as a teaser until we look at this example again in Chapter 4 (☞ 127).

Time of day, such as "9:17 am" or "12:30 pm"

Matching a time can be taken to varying levels of strictness. Something such as

⌈`[0-9]?[0-9]:[0-9][0-9]•(am|pm)`⌋

picks up both `9:17•am` and `12:30•pm`, but also allows `99:99•pm`.

Looking at the hour, we realize that if it is a two-digit number, the first digit must be a one. But ⌈1?[0-9]⌋ still allows an hour of 19 (and also an hour of 0), so maybe it is better to break the hour part into two possibilities: ⌈1[012]⌋ for two-digit hours and ⌈[1-9]⌋ for single-digit hours. The result is ⌈**(**1[012]**|**[1-9]**)**⌋.

The minute part is easier. The first digit should be ⌈[0-5]⌋. For the second, we can stick with the current ⌈[0-9]⌋. This gives ⌈(1[012]|[1-9]):[0-5][0-9]•(am|pm)⌋ when we put it all together.

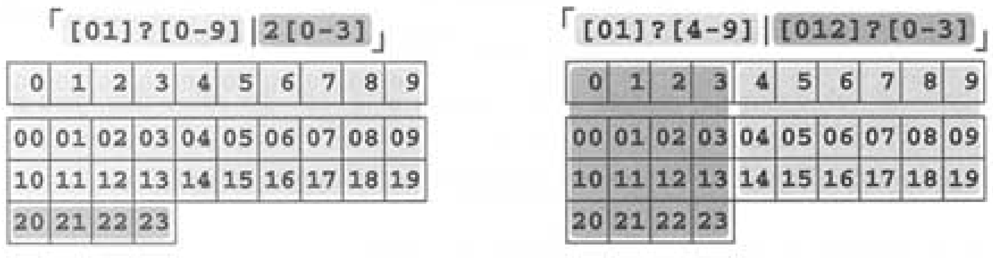
Using the same logic, can you extend this to handle 24-hour time with hours from 0 through 23? As a challenge, allow for a leading zero, at least through to 09:59. ❖ Try building your solution, then turn the page to check mine.

Extending the time regex to handle a 24-hour clock

❖ Answer to the question on page 23.

There are various solutions, but we can use similar logic as before. This time. I'll break the task into three groups: one for the morning (hours `00` through `09`, with the leading zero being optional), one for the daytime (hours `10` through `19`), and one for the evening (hours `20` through `23`). This can be rendered in a pretty straightforward way: ⌈`0?[0-9]|1[0-9]|2[0-3]`⌋.

Actually, we can combine the first two alternatives, resulting in the shorter ⌈`[01]?[0-9]|2[0-3]`⌋. You might need to think about it a bit to convince yourself that they'll really match exactly the same text, but they do. The figure below might help, and it shows another approach as well. The shaded groups represent numbers that can be matched by a single alternative.



Regular Expression Nomenclature

"Regex"

As you might guess, using the full phrase "regular expression" can get a bit tiring, particularly in writing. Instead, I normally use "regex". It just rolls right off the tongue (it rhymes with "FedEx") and it is amenable to a variety of uses like "when you regex . . ." "budding regexers," and even "regexification."* I use the phrase "regex engine" to refer to the part of a program that actually does the work of carrying out a match attempt.

"Matching "

When I say a regex *matches* a string, I really mean that it matches in a string. Technically, the regex ⌈`a`⌋ doesn't match `cat`, but matches the `a` *in* `cat`. It's not something that people tend to confuse, but it's still worthy of mention

---

* You might also come across 'regexp' I recommend eschewing this unpronounceable blot on English

"Metacharacter"

The concept of a character being a metacharacter (or "metasequence"—I use the words interchangeably) depends on exactly where in the regex it's used. For example, ⌈`*`⌋ is a metacharacter, but only when it's not within a character class and when not escaped. "Escaped" means that it has a backslash in front of it—Usually. The star is escaped in ⌈`\*`⌋ , but not in ⌈`\\*`⌋ (where the first backslash escapes the second), although the star "has a backslash in front of it" in both examples.

Depending upon the regex flavor, there are various situations when certain characters are and aren't metacharacters. Chapter 3 discusses this in more detail.

"Flavor"

As I've hinted, different tools use regular expressions for many different things, and the set of metacharacters and other features that each support can differ. Some might not support such-and-such a metacharacter, others might add this-and-that additional feature. Let's look at word boundaries again as an example. Some versions of *egrep* support the \<…\> notation we've seen. However, some do not support the separate word-start and word-end, but one catch-all ⌈`\b`⌋ metacharacter. Still others support both, and there are certainly those that support neither.

I use "flavor" to describe the sum total of all these little implementation decisions. In the language analogy, it's the same as a dialect of an individual speaker. Superficially, this refers to which metacharacters are and aren't supported, but there's much more to it. Even if two programs both support ⌈`\<…\>`⌋, they might disagree on exactly what they do and don't consider to be a word. This concern is important if you really want to *use* the tool. These kind of "behind-the-scenes" differences are the focus of Chapter 4.

Don't confuse "flavor" with "tool." Just as two people can speak the same dialect, two completely different programs can support exactly the same regex flavor. Also, two programs with the same name (and built to do the same task) often have slightly (and sometimes not-so-slightly) different flavors.

"Subexpression"

The term *subexpression* simply means any part of a larger expression, but usually refers to that part of an expression within parentheses, or an alternation alternative. For example, with ⌜`^(Subject|Date):•`⌟, the ⌜`Subject|Date`⌟ is usually referred to as a subexpression. Within that, the alternatives ⌜`Subject`⌟ and ⌜`Date`⌟ are referred to as subexpressions as well.

Something such as ⌜`1-6`⌟ isn't considered a subexpression of ⌜`H[1-6]•*`⌟, since the ⌜`1-6`⌟ is part of an unbreakable "unit,' the character class. Conversely, ⌜`H`⌟, ⌜`[1-6]`⌟, and ⌜`•*`⌟ are all subexpressions of the original.

Unlike alternation, quantifiers (star, plus, and friends) always work with the smallest immediately-preceding subexpression. This is why with 「`mis+pell`」, the + governs the 「`s`」, not the 「`mis`」 or 「`is`」. Of course, when what immediately precedes a quantifier is a parenthesized subexpression, the entire subexpression (no matter how complex) is taken as one unit.

"Character"

As I mentioned in an earlier footnote, *character* can be a loaded term in computing. The character that a byte represents is merely a matter of interpretation. A byte with such-and-such a value has that same value in any context in which you might wish to consider it, but which *character* that value represents depends on the encoding in which it is viewed. For example, two bytes with decimal values `64` and `53` represent the characters "@" and "5" if considered as ASCII, yet on the other hand are completely different if considered as EBCDIC (they are a space and the `<TRN>` character, whatever that is).

On the third hand, if considered in the JIS (ISO-2022-JP) encoding, those two bytes together represent the single character 正 (you might recognize this from the phrase mentioned in the "The Language Analogy" section on page 5). Yet, to represent that same 正 in the EUC-JP encoding requires two completely different bytes. Those bytes, by the way, yield the two characters "Àµ" in the *Latin-1* (ISO-8859-1) encoding, and the one Korean character 삵 in the Unicode encoding (but only from Version 2.0, mind you).*

You see what I mean. Regex tools generally treat their data as a bunch of bytes without regard to the encoding you might be intending. Searching for 「Àµ」 with most tools still finds 正 in EUC-JP-encoded data and 삵 in Unicode data.

These issues are immediate (and even more complex than I've led you to believe here) to someone working with data encoded in Unicode or any other multiple-byte encoding. However, these issues are irrelevant to most of you, so I use "byte" and "character" interchangeably.

Improving on the Status Quo

When it comes down to it, regular expressions are not difficult. But if you talk to the average user of a program or language that supports them, you will likely find someone that understands them "a bit," but does not feel secure enough to really use them for anything complex or with any tool but those they use most often.

---

* The definitive book on multiple-byte encodings is Ken Lunde's *Understanding Japanese Information Processing,* also published by O'Reilly & Associates. (Japanese title: 日本語情報処理). As I was going to press, Ken was working on his Second Edition, tentatively retitled *Understanding CJKV Information Processing.* The CJKV stands for *Chinese, Japanese, Korean,* and *Vietnamese,* which are languages that tend to require multiple-byte encodings.

Traditionally, regular expression documentation tends to be limited to a short and incomplete description of one or two metacharacters, followed by a table of the rest. Examples often use meaningless regular expressions like ⌈`a*((ab)*|b*)`⌋, and text like `'a·xxx·ce·xxxxxx·ci·xxx·d'`. They also tend to completely ignore subtle but important points, and often claim that their flavor is the same as some other well-known tool, almost always forgetting to mention the exceptions where they inevitably differ. The state of regex documentation needs help.

Now, I don't mean to imply that this chapter fills the gap. Rather, it merely provides the foundation upon which the rest of the book is built. It may be ambitious, but I hope this book will fill the gaps for you. Perhaps because the documentation has traditionally been so lacking, I feel the need to make the extra effort to make things really clear. Because I want to make sure you can use regular expressions to their fullest potential, I want to make sure you really, *really* understand them.

This is both good and bad.

It is good because you will learn how to *think* regular expressions. You will learn which differences and peculiarities to watch out for when faced with a new tool with a different flavor. You will know how to express yourself even with a weak, stripped-down regular expression flavor. When faced with a particularly complex task, you will know how to work through an expression the way the program would, constructing it as you go. In short, you will be comfortable using regular expressions to their fullest.

The problem is that the learning curve of this method can be rather steep:

- *How regular expressions are used*—Most programs use regular expressions in ways that are more complex than *egrep*. Before we can discuss in detail how to write a really useful expression, we need to look at the ways regular expressions can be used. We start in the next chapter.

- *Regular expression features*—Selecting the proper tool to use when faced with a problem seems to be half the battle, so I don't want to limit myself to only using one utility throughout the book. Different programs, and often even different versions of the same program, provide different features and metacharacters. We must survey the field before getting into the details of using them. This is Chapter 3.

- *How regular expressions really work*—Before we can learn from useful (but often complex) examples, we need to know just how the regular expression search is conducted. As we'll see, the order in which certain metacharacters are checked in can be very important. In fact, regular expressions can be implemented in different ways, so different programs sometimes do different things with the same expression. We examine this meaty subject in Chapters 4 and 5.

This last point is the most important and the most difficult to address. The discussion is unfortunately sometimes a bit dry, with the reader chomping at the bit to get to the fun part—tackling real problems. However, understanding how the regex engine really works is the key to *true understanding.*

You might argue that you don't want to be taught how a car works when you simply want to know how to drive. But learning to drive a car is a poor analogy in which to view regular expressions. My goal is to teach you how to solve problems with regular expressions, and that means constructing regular expressions. The better analogy is not how to drive a car, but how to build one. Before you can build a car, you have to know how it works.

Chapter 2 gives more experience with driving, Chapter 3 looks at the bodywork of a regex flavor, and Chapter 4 looks at the engine of a regex flavor. Chapter 3 also provides a light look at the history of driving, shedding light on why things are as they are today. Chapter 5 shows you how to tune up certain kinds of engines, and the chapters after that examine some specific makes and models. Particularly in Chapters 4 and 5, we'll spend a lot of time under the hood, so make sure to have your overalls and shop rags handy.

Summary

Table 1-3 summarizes the *egrep* metacharacters we've looked at in this chapter. In addition, be sure that you understand the following points:

- Not all *egrep* programs are the same. The supported set of metacharacters, as well as their meanings, are often different—see your local documentation.

- Three reasons for using parentheses are grouping, capturing, and constraining alternation.

- Character classes are special: rules about metacharacters are entirely different within character classes.

- Alternation and character classes are fundamentally different, providing unrelated services that appear, in only one limited situation, to overlap.

- A negated character class is still a "positive assertion"—even negated, a character class must match a character to be successful. Because the listing of characters to match is negated, the matched character must be one of those *not* listed in the class.

- The useful `-i` option discounts capitalization during a match.

Table 1- 3: Egrep Metacharacter Summary

Metacharacter		Matches
Items to Match a Single Character —		
.	*dot*	Matches any one character
[...]	*character class*	Matches any character listed
[^...]	*negated character class*	Matches any character not listed
\char	*escaped character*	When *char* is a metacharacter, or the escaped combination is not otherwise special, matches the literal *char*
Items Appended to Provide "Counting": The Quantifiers —		
?	*question*	One allowed, but is optional
*	*star*	Any number allowed, but are optional
+	*plus*	One required, additional are optional
{*min*, *max*}	*specified range**	*Min* required, *max* allowed
Items That Match Positions —		
^	*caret*	Matches the position at the start of the line
$	*dollar*	Matches the position at the end of the line
\<	*word boundary**	Matches the position at the start of a word
\>	*word boundary**	Matches the postion at the end of a word
Other —		
|	alternation	Matches either expression it separates
(...)	*parentheses*	Limits scope of alternation, provides grouping for the quantifiers, and "captures" for backreferences
\1, \2, ...	*backreference**	Matches text previously matches within first, second, etc., set of parentheses.

* *not supported by all versions of egrep*

- There are three types of escaped items:

1. The pairing of ⌈\⌋ and a metacharacter is a metasequence to match the literal character (for example, ⌈*⌋ matches a literal asterisk).

2. The pairing of ⌈\⌋ and selected non-metacharacters becomes a metasequence with an implementation-defined meaning (for example, ⌈\<⌋ often means "start of word").

3. The pairing of ⌈\⌋ and any other character defaults to simply matching the character (that is, the backslash is ignored).

Remember, though, that a backslash within a character class is not special at all, so it provides no "escape services" in such a situation.

- Items governed by a question mark or star don't need to actually match any characters to "match successfully." They are *always* successful, even if they don't match anything.

Personal Glimpses

The doubled-word task at the start of this chapter might seem daunting, yet regular expressions are so powerful that we could solve much of the problem with a tool as limited as *egrep,* right here in the first chapter. I'd like to fill this chapter with flashy examples, but because I've concentrated on the solid foundation for the later chapters, I fear that someone completely new to regular expressions might read this chapter, complete with all the warnings and cautions and rules and such, and feel "why bother?"

Recently, my brothers were teaching some friends how to play *schafkopf,* a card game that's been in my family for generations. It is much more exciting than it appears at first glance, but has a rather steep learning curve. After about half an hour, my sister-in-law Liz, normally the quintessence of patience, got frustrated with the seemingly complex rules and said "can't we just play rummy?" Yet as it turned out, they ended up playing late into the night. Once they were able to get over the initial hump of the learning curve, a first-hand taste of the excitement was all it took to hook them. My brothers knew it would, but it took some time and work to get to the point where Liz and the others new to the game could appreciate what they were getting into.

It might take some time to become acclimated to regular expressions, so until you get a real taste of the excitement by using them to solve *your* problems, it might all feel just a bit too academic. If so, I hope you will resist the desire to "play rummy." Once you understand the power that regular expressions provide, the small amount of work spent learning them will feel trivial indeed.

2

Extended Introductory Examples

In this chapter:
- ***About the Examples***
- ***Matching Text with Regular Expressions***
- ***Modifying Text with Regular Expressions***

Remember the double-word problem from the first chapter? I said that a full solution could be written in just a few lines in a language like Perl. Such a solution might look like:

```
    $/ = ".\n";
    while (<>) {
       next if
!s/\b([a-z]+)((\s|<[^>]+>)+)(\l\b)/\e[7m$1\e[m$2\e[7m$4\e[m/ig;
       s/^([^\e]*\n)+//mg;      # Remove any unmarked lines.
       s/^/$ARGV: /mg;          # Ensure lines begin with filename.
       print;
     }
```

Yup, that's the *whole* program.

I don't expect you to understand it (*yet!*). Rather, I wanted to show an example beyond what *egrep* can allow, and to whet your appetite for the real power of regular expressions—most of this program's work is centered around its three regular expressions:

```
     \b([a-z]+)((\s|<[^>]+>)+)(\1\b)
     ^([^\e]*\n)+
     ^
```

That last ⌈`^`⌋ is certainly recognizable, but the other expressions have items unfamiliar to our *egrep*-only experiences (although ⌈`\b`⌋, at least, was mentioned briefly on page 25 as sometimes being a *word-boundary*, and that's what is it is here). This is because Perl's regex flavor is not the same as *egrep's.* Some of the notations are different, and Perl provides a much richer set of metacharacters. We'll see examples throughout this chapter.

About the Examples

Perl allows you to employ regular expressions in much more complex ways than *egrep.* Examples in Perl will allow us to see additional examples of regular expressions, but more importantly, will allow you to view them in quite a different context than with *egrep.* In the meantime, we'll be exposed to a similar (yet somewhat different) flavor of regular expression.

This chapter takes a few sample problems—validating user input; working with email headers—and wanders through the regular expression landscape with them. We'll see a bit of Perl, and insights into some thought processes that go into crafting a regex. During our journey, we will take plenty of side trips to look at various other important concepts as well.

There's nothing particularly special about Perl in this respect. I could just as easily use any number of other advanced languages (such as Tcl, Python, even GNU Emacs' *elisp*). I choose Perl primarily because it has the most ingrained regex support among these popular languages and is, perhaps, the most readily available. Also, Perl provides many other concise data-handling constructs that will alleviate much of the "dirty work," letting us concentrate on regular expressions. Just to quickly demonstrate some of these powers, recall the file-check example from page 2. The utility I used was Perl, and the command was:

```
    % perl -One 'print "$ARGV\n" if s/ResetSize//ig !=
s/SetSize//ig' *
```

(I don't expect that you understand this yet—I hope merely that you'll be impressed with the shortness of the solution.)

I like Perl, but it's important to not get too caught up in its trappings here. Remember, this chapter concentrates on *regular expressions.* As an analogy, consider the words of a computer science professor in a first-year course: "You're going to learn CS concepts here, but we'll use Pascal to show you." (Pascal is a traditional programming language originally designed for teaching.)*

Since this chapter doesn't assume that you know Perl, I'll be sure to introduce enough to make the examples understandable. (Chapter 7, which looks at all the nitty-gritty details of Perl, does assume some basic knowledge.) Even if you have experience with a variety of programming languages, Perl will probably seem quite odd at first glance because its syntax is very compact and its semantics thick. So, while not "bad," the examples are not the best models of The Perl Way of programming. In the interest of clarity, I'll not take advantage of much that Perl has to offer; I'll attempt to present programs in a more generic, almost pseudo-code style. But we *will* see some great uses of regular expressions.

---

* Thanks to William F. Maton, and his professor, for the analogy.

A Short Introduction to Perl

Perl is a powerful scripting language built by Larry Wall in the late 1980s, drawing ideas from many other programming languages. Many of its concepts of text handling and regular expressions are derived from two languages called *awk* and *sed,* both of which are quite different from a "traditional" language such as C or Pascal. Perl is available for many platforms, including DOS/Windows, MacOS, OS/2, VMS, and Unix. It has a powerful bent toward text-handling and is a particularly common tool used for World Wide Web CGIs (the programs that construct and send out dynamic Web pages). See Appendix A for information on how to get a copy of Perl for your system. I'm writing as of Perl Version 5.003, but the examples in this chapter are written to work with Version 4.036 or later.*

Let's look at a simple example:

```
$celsius = 30;
$fahrenheit = ($celsius * 9 / 5) + 32;  # calculate Fahrenheit
print "$celsius C is $fahrenheit F.\n"; # report both temps
```

When executed, this produces:

```
30 C is 86 F.
```

Simple variables, such as `$fahrenheit` and `$celsius`, always begin with a dollar sign and can hold a number or any amount of text. (In this example, only numbers are used.) Comments begin with # and continue for the rest of the line. If you're used to traditional programming languages like C or Pascal, perhaps most surprising is that variables can appear within Perl doublequoted strings. With the string `"$celsius C is $fahrenheit F.\n"`, each variable is replaced by its value. In this case, the resulting string is then printed. (The `\n` represents a newline.)

There are control structures similar to other popular languages:

```
$celsius = 20;
while ($celsius <= 45)
{
   $fahrenheit = ($celsius * 9 / 5) + 32; # calculateFahrenheit
   print "$celsius C is $fahrenheit F.\n";
   $celsius = $celsius + 5;
}
```

The body of the code controlled by the while loop is executed repeatedly so long as the condition (the **`$celsius <= 45`** in this case) is true. Putting this into a file, say *temps*, we can run it directly from the command line.

---

* Although all the examples in this chapter will work with earlier versions, as a general rule I *strongly* recommend using Perl version 5.002 or later. In particular, I recommend that the archaic version 4.036 not be used unless you have a very specific need for it.

Here's how it looks:

```
% perl -w temps
20 C is 68 F.
25 C is 77 F.
30 C is 86 F.
35 C is 95 F.
40 C is 104 F.
45 C is 113 F.
```

The `-w` option is neither necessary, nor has anything directly to do with regular expressions. It tells Perl to check your program more carefully and issue warnings about items it thinks to be dubious (such as using uninitialized variables and the like variables do not need to be predeclared in Perl). I use it here merely because it is good practice to always do so.

Matching Text with Regular Expressions

Perl uses regexes in many ways, the most simple being to check if a regex can match the text held in a variable. The following program snippet checks the string in variable `$reply` and reports whether it contains only digits:

```
if ($reply =~ m/^[0-9]+$/) {
    print "only digits\n";
} else {
    print "not only digits\n";
}
```

The mechanics of the first line might seem a bit strange. The regular expression is ⌈`^[0-9]+$`⌋, while the surrounding `m/…/` tells Perl what to do with it. The `m` means to attempt a *regular expression match*, while the slashes delimit the regex itself. The `=~` links `m/…/` with the string to be searched, in this case the contents of the variable `$reply`.

Don't confuse `=~` with `=` or `==`, as they are quite different. The operator `==` tests whether two numbers are the same. (The operator `eq`, as we will soon see, is used to test whether two *strings* are the same.) The = operator is used to assign a value to a variable, as with **`$celsius = 20`**. Finally, `=~` links a regex search with the target string to be searched. (In the example, the search is `m/^[0-9]+$/` and the target is `$reply`.) It might be convenient to read `=~` as "matches," such that

```
if ($reply =~ m/^[0-9]+$/) {
```

becomes:

> if the text in the variable reply matches the regex ⌜`^[0-9]+$`⌟, then . . .

The whole result of **`$reply =~ m/^[0-9]+$/`** is a *true* value if the ⌜`^[0-9]+$`⌟ matches the string in `$reply`, a *false* value otherwise. The if uses this true or false value to decide which message to print.

Note that a test such as **`$reply =~ m/[0-9]+/`** (the same as before except the wrapping caret and dollar have been removed) would be true if `$reply` contained at least one digit anywhere. The ⌈`^…$`⌋ ensures that `$reply` contains *only* digits.

Let's combine the last two examples. We'll prompt the user to enter a value, accept that value, then verify its form with a regular expression to make sure it's a number. If it is, we calculate and display the Fahrenheit equivalent. Otherwise, we issue a warning message.

```
    print "Enter a temperature in Celsius:\n";
    $celsius = <STDIN>;     #  this will read one line from the user
    chop ($celsius);        #  this removes the ending new line from $celsius

    if ($celsius =~ m/^[0-9]+$/) {
        $fahrenheit = ($celsius * 9 / 5) + 32; #  calculate Fahrenheit
        print "$celsius C = $fahrenheit F\n";
    } else {
        print "Expecting a number, so don't understand
\"$celsius\".\n";
    }
```

Notice in the last `print` how we escaped the quotes to be printed. This is similar to escaping a metacharacter in a regex. The section "metacharacters galore," in a few pages (☞ 41), discusses this in a bit more detail.

If we put this program into the file *c2f* we might run it and see:

```
    % perl -w c2f
    Enter a temperature in Celsius:
    22
    22 C = 71.599999999999994316 F
```

Oops. As it turns out, Perl's simple `print` isn't so good when it comes to floating-point numbers. I don't want to get bogged describing all the details of Perl in this chapter, so I'll just say without further comment that you can use `printf` ("print formatted") to make this look better (`printf` is similar to the `C` language's `printf`, or the format of Pascal, Tcl, *elisp*, and Python):

```
    printf "%.2f C = %.2f F\n", $celsius, $fahrenheit;
```

This doesn't change the values of the variables, but merely their display. The result now looks like

```
Enter a temperature in Celsius:
22
22.00 C = 71.60 F
```

which is much nicer.

Toward a More Real-World Example

I think it'd be useful to extend this example to allow negative and fractional temperature values. The math part of the program is fine Perl normally makes no distinction between integers and floating-point numbers. We do, however, need to modify the regex to let negative and floating-point values pass. We can insert a leading 「`-?`」 to allow a leading minus sign. In fact, we may as well make that 「`[-+]?`」 to allow a leading plus sign, too.

To allow an optional decimal part, we add 「`(\.[0-9]*)?`」. The escaped dot matches a literal period, so 「`\.[0-9]*`」 is used to match a period followed by any number of optional digits. Since 「`\.[0-9]*`」 is enclosed by 「`(…)?`」, it as a whole becomes optional. (This is logically different from 「`\.?[0-9]*`」, which allows additional digits to match even if 「`\.`」 does not match.)

Putting this all together, we get

```
     if ($celsius =~ m/^[-+]?[0-9]+(\.[0-9]*)?$/)  {
```

as our check line. It allows numbers such as `32`, `-3.723`, and `+98.6`. It is actually not quite perfect: it doesn't allow a number that begins with a decimal point (such as `.357`). Of course, the user can just add a leading zero (i.e., `0.357`) to allow it to match, so I don't consider it a major shortcoming. This floating-point problem can have some interesting twists, and I look at it in detail in Chapter 4 (☞ 127).

Side Effects of a Successful Match

Let's extend the example further to allow someone to enter a value in either Fahrenheit or Celsius. We'll have the user append a `C` or `F` to the temperature entered. To let this pass our regular expression, we can simply add 「`[CF]`」 after the expression to match a number, but we still need to change the rest of the program to recognize which kind of temperature was entered and to compute the other.

Perl, like many regex-endowed languages, has a useful set of special variables that refer to the text matched by parenthesized subexpressions in a regex. In the first chapter, we saw how some versions of *egrep* support ⌈`\1`⌋, ⌈`\2`⌋, ⌈`\3`⌈, etc. as metacharacters within the regex itself. Perl supports these, and also provides variables which can be accessed outside of the regular expression, after a match has been completed. These Perl variables are `$1`, `$2`, `$3`, etc. As odd as it might seem, these *are* variables. The variable names just happen to be numbers. Perl sets them every time a regular expression is successful. Again, use the metacharacter ⌈`\1`⌋ within the regular expression to refer to some text matched earlier during the same match attempt, and use the variable `$1` to allow the program to refer to the text once the match has been successfully completed.

To keep the example uncluttered and focus on what's new, I'll remove the fractional-value part of the regex for now, but we'll return to it again soon. So, to see `$1` in action, compare:

```
$celsius =~ m/^[-+]?[0-9]+[CF]$/
$celsius =~ m/^([-+]?[0-9]+)([CF])$/
```

Do the added parentheses change the meaning of the expression? Well, to answer that, we need to know whether they:

- provide grouping for star or other quantifiers?
- provide an enclosure for 「|」?

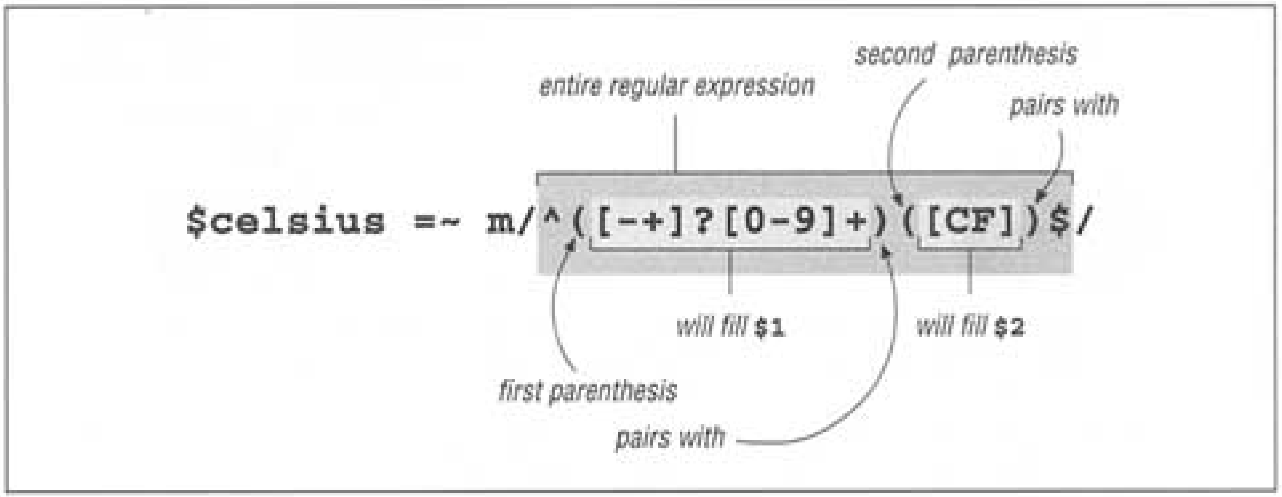

second parenthesis
entire regular expression
pairs with
$celsius =~ m/^([-+]?[0-9]+)([CF])$/
will fill $1
will fill $2
first parenthesis
pairs with

Figure 2-1:
capturing parentheses

The answer is no on both counts, so what is matched remains unchanged. However, they do enclose two subexpressions that match "interesting" parts of the string we are checking. As Figure 2-1 illustrates, `$1` will hold the number entered and `$2` will hold the `C` or `F` entered. Referring to the flowchart in Figure 2-2 on the next page, we see that this allows us to easily decide how to proceed after the match.

Assuming the program shown on the next page is named *convert*, we can use it like this:

```
% perl -w convert
Enter a temperature (i.e. 32F, 100C):
39F
3.89 C = 39.00 F
% perl -w convert
Enter a temperature (i.e. 32F, 100C):
39C
39.00 C = 102.20 F
% perl -w convert
Enter a temperature (i.e. 32F, 100C):
oops
Expecting a number, so don't understand "oops.
```

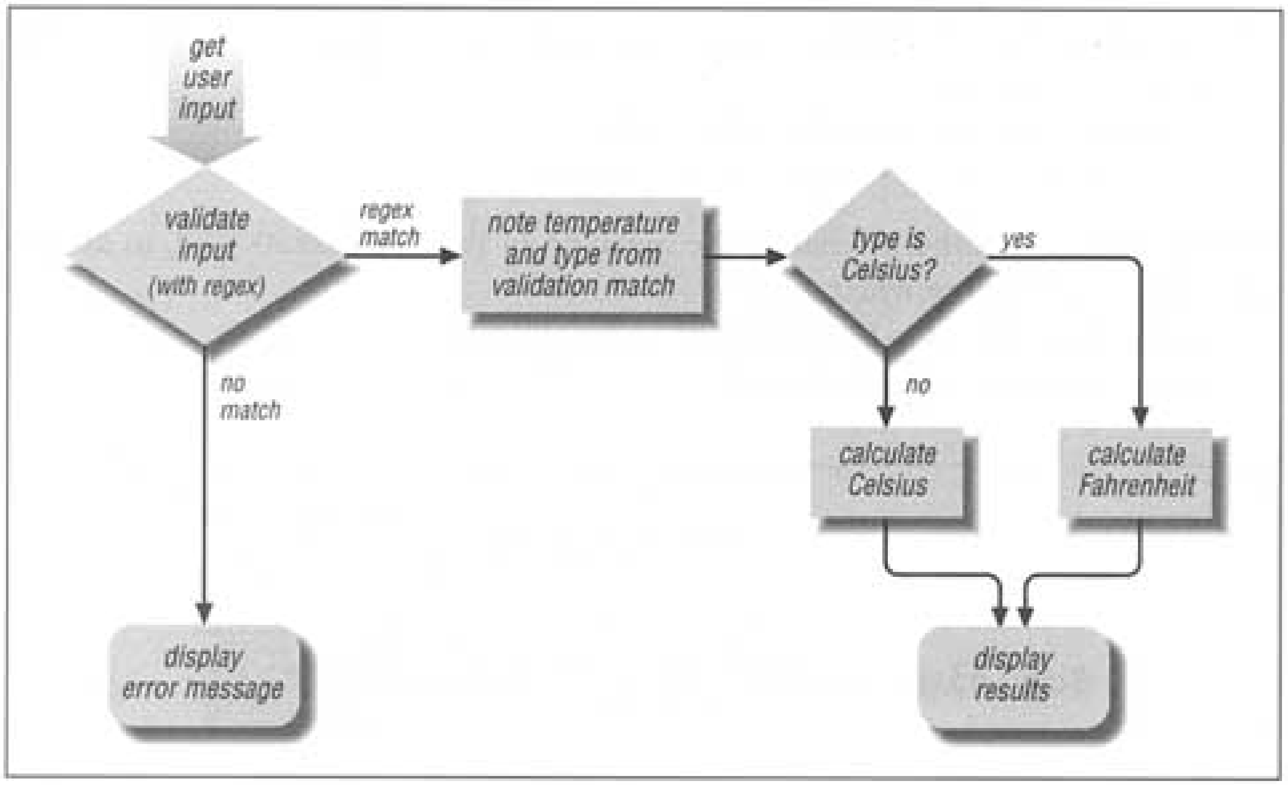

get
user
input
validate
input
(with regex)
regex
match
note temperature
and type from
validation match
type is
Celsius?
yes
no
no
match
calculate
Celsius
calculate
Fahrenheit
display
error message
display
results

Figure 2-2:
Temperature-conversion program's logic flow

```
print "Enter a temperature (i.e. 32F, 100C):\n";
$input = <STDIN>; #  this will read one line from the user
chop ($input);    #  this removes the ending new line from $input

if ($input =~ m/^([-+]?[0-9]+)([CF])$/)
{
    #  If we get in here, we had a match. $1 is the number, $2 is "C" or "F".
    $InputNum = $1;  #  save to named variables to make the . . .
    $type     = $2;  #  . . . rest of the program easier to read.

    if ($type eq "C") {     #  'eq' tests if two strings are equal
        #  The input was Celsius, so calculate Fahrenheit
        $celsius = $InputNum;
        $fahrenheit = ($celsius * 9 / 5) + 32;
    } else {
        #  Gee, must be an "F" then, so calculate Celsius
        $fahrenheit = $InputNum;
        $celsius = ($fahrenheit - 32) * 5 / 9;
    }
    #  at this point we have both temperatures, so display the results:
        printf "%.2f C = %.2f F\n", $celsius, $fahrenheit;
    } else {
        # The initial regex did not match, so issue a warning.
        print "Expecting a number, so don't understand
\"$input\".\n";
    }
```

Negated matches

Our program logic has the structure:

```
    if ( logical test ) {
          ⋮
       . . . LOTS OF PROCESSING if the test result was true . . .
          ⋮
    } else {
       . . . just a bit of processing if the test result was false . . .
    }
```

Any student of structured programming knows (or should know) that when one branch of an `if` is short and the other long, it is better to put the short one first if reasonably possible. This keeps the `else` close to the `if`, which makes reading much easier.

To do this with our program, we need to invert the sense of the test. The "when it doesn't match" part is the short one, so we want our test to be true when the regex doesn't match. One way to do this is by using `!~` instead of `=~`, as shown in:

```
    $input !~ m/^([-+]?[0-9]+)([CF])$/
```

The regular expression and string checked are the same. The only difference is that the result of the whole combination is now *false* if the regex *matches* the string, true otherwise. When there is a match, `$1`, `$2`, etc., are still set. Thus, that part of our program can now look like:

```
    if ($input !~ m/^([-+]?[0-9]+)([CF])$/) {
        print "Expecting a number, so don't understand
\"$input\" .\n";
    } else {
       If we get in here, we had a match. $1 is the number, $2 is "C" or "F".
       ⋮
    }
```

Intertwined Regular Expressions

With advanced programming languages like Perl, regex use can become quite intertwined with the logic of the rest of the program. For example let's make three useful changes to our program: allow floating-point numbers as we did earlier, allow a lowercase `f` or `c` to be entered, and allow spaces between the number and letter. Once all these changes are done, input such as '`98.6·f`' will be allowed.

Earlier, we saw how we can allow floating-point numbers by adding ⌈`(\.[0-9]*)?`⌋ to the expression:

```
    if ($input =~  m/^([-+]?[0-9]+(\.[0-9]*)?)([CF])$/)
```

Notice that it is added *inside* the first set of parentheses. Since we use that first set to capture the number to compute, we want to make sure that they will capture the fractional portion as well. However, the added set of parentheses, even though ostensibly used only to group for the question mark, also has the side effect of capturing into a variable. Since the opening parenthesis of the pair is the second (from the left), it captures into `$2`. This is illustrated in Figure 2-3.

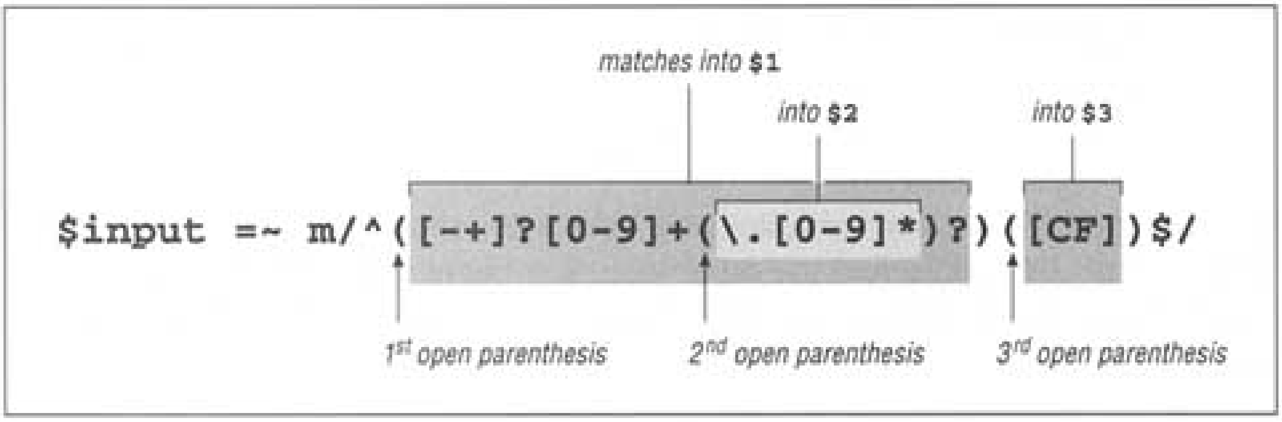


Figure 2-3:
Nesting parentheses

You can see how closing parentheses nest with opening ones. Adding a set of parentheses earlier in the expression doesn't influence the meaning of ⌈`[CF]`⌋ directly, but it does so indirectly because the parentheses surrounding it have become the third pair. Becoming the third pair means that we need to change the assignment to `$type` to refer to `$3` instead of `$2`.

Allowing spaces between the number and letter is not so involved. We know that an unadorned space in a regex requires exactly one space in the matched text, so ⌈●*⌋ can be used to allow any number of spaces (but still not require any):

```
if ($input =~ m/^([-+]?[0-9]+(\.[0-9]*)?)_*([CF])$/)
```

This provides for a certain amount of flexibility, but since we are trying to make something useful in the real world, let's construct the regex to also allow for other kinds of *whitespace* as well. Tabs, for instance, are quite common. Writing ⌈TAB*⌋; of course, doesn't allow for spaces, so we need to construct a character class to match either: ⌈[● TAB] *⌋,. Here's a quick quiz: how is this fundamentally different from ⌈(*|TAB*)⌋?

❖ After considering this, turn the page to check your answer.

In this book, spaces and tabs are easy to notice because of the • and TAB typesetting conventions I've used. Unfortunately, it is not so on-screen. If you see something like `[    ]*`, you can guess that it is probably a space and a tab, but you can't be sure until you check. For convenience, Perl regular expressions provide the ⌈`\t`⌋ metacharacter. It simply matches a tab—its only benefit over a literal tab is that it is visually apparent, so I use it in my expressions. Thus, ⌈`[•TAB]*`⌋ becomes ⌈`[•\t]*`⌋.

Some other convenience metacharacters are ⌈`\n`⌋ (newline), ⌈`\f`⌋ (ASCII formfeed), and ⌈`\b`⌋ (backspace). *Wait a moment!* Earlier I said that ⌈`\b`⌋ was for matching a word boundary. *So, which is it?* Well, actually, it's both!

A short aside—metacharacters galore

We saw `\n` in earlier examples, but in those cases it was in a string, not a regular expression. Perl *strings* have metacharacters of their own, and these are completely distinct from *regular expression* metacharacters. It is a common mistake for new programmers to get them confused.

However, as it turns out, some of the string metacharacters conveniently look the same as some comparable regex metacharacters. You can use the string metacharacter `\t` to get a tab into your string, while you can use the regex metacharacter ⌈`\t`⌋ to insert a tab-matching element into your regex.

The similarity is convenient, but I can't stress enough how important it is to maintain the distinction between the different types of metacharacters. For such a simple example as `\t` it doesn't seem important, but as we'll later see when looking at numerous different languages and tools, knowing which metacharacters are being used in each situation is extremely important.

In fact, we have already seen multiple sets of metacharacters conflict. In Chapter 1, while working with *egrep,* we generally wrapped our regular expressions in single quotes. The whole command line is written at the command-shell prompt, and the shell recognizes several of its own metacharacters. For example, to the shell, the space is a metacharacter that delimits the command from the arguments and the arguments from each other. With many shells, singlequotes are metacharacters that tell the shell to not recognize other shell metacharacters in the text between the quotes. (DOS uses doublequotes.)

Using the quotes for the shell allows us to use spaces in our regular expression. Without the quotes, the shell would interpret the spaces in its own way instead of passing them through to *egrep* to interpret in *its* way. Many shells also recognize metacharacters such as `$`, `*`, `?`, and so on—characters that we are likely to want to use in a regex.

Now, all this talk about other shell metacharacters and Perl's string metacharacters has nothing to do with regular expressions themselves, but it has everything to do with *using* regular expressions in real situations, and as we move through this book we will see numerous and sometimes complex situations where we need to take advantage of multiple levels of simultaneously interacting metacharacters.

And what about this ⌜`\b`⌟ business? This *is* a regex thing: In Perl, it normally matches a word boundary, but within a character class it matches a backspace. A word boundary would make no sense as part of a class, so Perl is free to let it mean something else. The warnings in the first chapter about how a character class's "sub language" is different from the main regex language certainly apply to Perl (and every other regex flavor as well).

How do ⌈ *[•TAB] **⌋ and ⌈ *(•*|TAB*)* ⌋ compare?

✣ Answer to the question on page 40.

⌈`(•*|TAB*)`⌋ allows either ⌈`TAB*`⌋ or ⌈`TAB*`⌋ to match, which allows either some spaces (or nothing) or some tabs (or nothing). It doesn't, however, allow a *combination* of spaces and tabs.

⌈`[•TAB]*`⌋ matches ⌈`[•TAB]`⌋ any number of times. With a string such as `'TAB••'` itmatches three times, a tab the first time and spaces the rest.

⌈`[•TAB]*`⌋ is logically equivalent to ⌈`(•|TAB)*`⌋, although for reasons shown inChapter 4, a character class is often much more efficient.

Generic "whitespace" with `\s`

While discussing whitespace, we left off with ⌈`[•\t] *`⌋. This is fine, but Perl regular expressions again provide a useful shorthand. Different from ⌈`\t`⌋, which simply represents a literal tab, the metacharacter ⌈`\s`⌋ is a shorthand for a whole character class that matches any "whitespace character." This includes (among others) space, tab, newline, and carriage return. With our example, the newline and carriage return don't really matter one way or the other, but typing ⌈`\s*`⌋ is easier than ⌈`[•\t]*`⌋. After a while you get used to seeing it, and ⌈`\s*`⌋ becomes easy to read even in complex regular expressions.

Our test now looks like: `$input =~ m/^([-+]?[0-9]+(\.[0-9]*)?)\s*([CF])$/`

Lastly, we want to allow a lowercase letter as well as uppercase. This is as easy as adding the lowercase letters to the class: ⌈`[CFcf]`⌋. However, I'd like to show another way as well: `$input =~ m/^([-+]?[0-9]+(\.[0-9]*)?)\s*([CF])$/i`

The `i` is a *modifier*, and placing it after the `m/…/` instructs Perl to do the match in a case-insensitive manner. The `i` is not part of the regex, but part of the `m/…/` syntactic packaging that tells Perl *what* you want to do with exactly *which* regex. It's a bit too cumbersome to say "the `i` modifier" all the time, so normally `"/i"` is used (even though you don't add an extra / when actually using it). We will see other modifiers as we move along, including `/g` later in this chapter.

Let's try our new program:

```
% perl -w convert
Enter a temperature (i.e. 32F, 100C):
32 f
0.00 C = 32.00 F
% perl -w convert
Enter a temperature (i.e. 32F, 100C):
50 c
10.00 C = 50.00 F
```

Oops! Did you notice that in the second try we thought we were entering 50° Celsius, yet it was interpreted as 50° Fahrenheit? Looking at the program's logic, do you see why? Let's look at that part of the program again:

```
    if ($input =~ m/^([-+]?[0-9]+(\.[0-9]*)?)\s*([CF])$/i)
    {
        ⋮
        $type = $3;  #  using $type instead of $3 makes the rest of the program more readable

        if ($type eq "C") {   #   'eq' tests if two strings are equal
            ⋮
        } else {
            ⋮
```

Although we modified the regex to allow a lowercase `f`, we didn't update the rest of the program to respond to it appropriately. As it is now, if `$type` isn't exactly `C`, we assume the user entered Fahrenheit. Since we now also allow `c` to mean Celsius, we need to update the `type` test:*

```
    if ($type eq "C" or $type eq "c")  {
```

Now things work as we want. These examples show how the use of regular expressions can become intertwined with the rest of the program.

It is interesting to note that the check for c could just as easily be done with a regex. How would you do that? ❖ Turn the page to check your answer.

Intermission

Although we have spent much of this chapter coming up to speed with Perl, there are a number of regex points to summarize:

- Perl regular expressions are different from *egrep* regular expressions; most tools have their own particular flavor of regular expressions. Perl's appear to be of the same general type as *egrep's*, but have a richer set of metacharacters.

- Perl can check a string in a variable against a regex using the construct **`$variable =~ m/`**…**`/.`** The `m` indicates a *match* is requested, while the slashes delimit (and are not part of) the regular expression. The whole test, as a unit, is either true or false.

- The concept of metacharacters—characters with special interpretations—is not unique to regular expressions. As discussed earlier about shells and doublequoted strings, multiple contexts often vie for interpretation. Knowing the various contexts (shell, regex, string, among others), their metacharacters, and how they can interact becomes more important as you learn and use Perl, Tcl, GNU Emacs, *awk*, Python, or other advanced scripting languages.

---

* Older versions of Perl use **||** instead of with this snippet.

One way to check a variable for a single letter

❖ *Answer to the question on page 43.*

The check for **`c`** can be done with **`$type =~ m/^[cC]$/`**. In fact, since we know for sure that `$type` will contain nothing more or less than exactly one letter, we can even omit the ⌈`^`…`$`⌋ in this case. While we're at it, we could even use **`$type =~ m/c/i`**, which lets `/i` do some of the work for us.

Since it is so easy to check directly for `c` and `C`, it seems to be overkill to use a regular expression. However, were the problem to allow for any mixture of case for the whole word "celsius," using a regular expression instead of checking directly for `celsius`, `Celsius`, `CeLsIuS`, and so on (128 possible combinations!) would make a lot of sense.

As you learn more Perl, you might find yet other approaches that are better still, such as perhaps **`lc($type) eq "celsius"`**.

- Among the more useful shorthands that Perl regexes provide (some of which we haven't seen yet):

`\t`	a tab character
`\n`	a newline character
`\r`	a carriage-return character
`\s`	a class to match any "whitespace" character (space, tab, newline, formfeed, and so on)
`\s`	anything not ⌈`\s`⌋
`\w`	⌈`[a-zA-Z0-9_]`⌋ (useful as in ⌈`\w+`⌋, ostensibly to match a word)
`\w`	anything not ⌈`\w`⌋, i.e., ⌈`[^a-zA-Z0-9_]`⌋
`\d`	⌈`[0-9]`⌋, i.e., a digit
`\D`	anything not ⌈`\d`⌋, i.e., ⌈`[^0-9]`⌋

- The `/i` modifier makes the test case-insensitive. Although written in prose as `"/i"`, only `"i"` is actually appended after the closing delimiter.

- After a successful match, the variables `$1`, `$2`, `$3`, etc., are available and hold the text matched by their respective parenthesized subexpressions in the regex. Subexpressions are numbered by counting open parentheses from the left, starting with one. The subexpressions can be nested, such as with ⌈`(United•States(•of•America)?)`⌋.

    Parentheses can be intended for grouping only, but as a byproduct they still match into one of the special variables. On the other hand, they are often inserted specifically to fill a numbered variable. This technique allows you to use a regex to pluck information from a string.

Modifying Text with Regular Expressions

So far, the examples have centered on finding, and at times, "plucking out" information from a string. Now we will take a look at *substitution,* or *search and replace,* a regex feature that Perl and many tools offer.

As we have seen, **`$var =~ m/regex/`** attempts to match the given regular expression to the text in the given variable, and returns true or false appropriately. The similar construct **`$var =~ s/regex/replacement/`** takes it a step further: if the regex is able to match the string in `$var`, the text actually matched will be replaced by *replacement.* The regex is the same as with `m/…/`, but the replacement (between the middle and final slash) is treated the same as a doublequoted string. This means that you can include references to variables (including, as is particularly useful, `$1`, `$2`, and so on to refer to parts of what was just matched).

Thus, with **`$var =~ s/…/…/`** the value of the variable is actually changed (but if there is no match to begin with, no replacement is made and the variable is left unchanged). For example, if `$var` contained `Jeff·Friedl` and we ran

```
$var =~ s/Jeff/Jeffrey/;
```

`$var` would end up with `Jeffrey·Friedl`. But if `$var` held `Jeffrey·Friedl` to begin with, it would end up with `Jeffreyrey·Friedl` after running the snippet. Perhaps we should use a word-boundary metacharacter. As mentioned in the first chapter, some versions of *egrep* support ⌈`\<`⌋ and ⌈`\>`⌋ for their *start-of-word* and *end-of-word* metacharacters. Perl, however, provides the catch-all ⌈`\b`⌋ to match either:

```
$var =~ s/\bJeff\b/Jeffrey/;
```

Here's a slightly tricky quiz: like `m/…/`, the `s/…/…/` operation can use modifiers such as `/i`. Practically speaking, what does

```
$var =~ s/\bJeff\b/Jeff/i;
```

accomplish? ❖ Flip the page to check your answer.

Let's look at rather a humorous example that shows the use of a variable in the replacement string. I can imagine a form-letter system that might use the following as the basic letter text:

```
Dear <FIRST>,
You have been chosen to win a brand new <TRINKET>! Free!
Could you use another <TRINKET> in the <FAMILY> household?
Yes <SUCKER>, I bet you could! Just respond by.....
```

To process this for a particular recipient, you might have the program load:

```
$given = 'Tom';
$family = 'Cruise';
$wunderprize = '100% genuine faux diamond';
```

Just what does `$var = s/\bJeff\b/Jeff/i` ***do?***

❖ Answer to the question on page 45.

It might be tricky because of the way I posed it. Had I used ⌜`\bJEFF\b`⌟ or ⌜`\bjeff\b`⌟ or perhaps ⌜`\bjEfF\b`⌟ as the regex, the intent might have been more obvious. Because of `/i`, the word "Jeff" will be found without regard to capitalization. It will then be replaced by `'Jeff'`, which has exactly the capitalization you see. (`/i` has no effect on the replacement text, although there are other modifiers examined in Chapter 7 that do.)

Once setup, you could then "fill out the form" with:

```
    $letter =~ s/<FIRST>/$given/g;
    $letter =~ s/<FAMILY>/$family/g;
    $letter =~ s/<SUCKER>/$given $family/g;
    $letter =~ s/<TRINKET>/fabulous $wunderprize/g;
```

Each substitution's regex looks for a simple marker, and when found, replaces it with the text wanted in the final message. In the first two cases, the text comes from simple variable references (similar to having nothing more than a variable in a string, i.e., `"$given"`). The third line replaces the matched text with the equivalent of "`$given $family`", and the fourth with `"fabulous $wunderprize"`. If you just had the one letter, you could skip these variables and use the desired text directly, but this method makes automation possible, such as when reading names from a list.

We haven't seen the `/g` "global match" modifier yet. It instructs the `s/…/…/` to continue trying to find more matches (and make more replacements) after (and where) the first substitution completes. This is needed if we want each substitution to do all the replacements it can, not just one.

The results are predictable, but rather funny in a way that I've seen all too often:

> Dear Tom,
> You have been chosen to win a brand new fabulous 100% genuine faux diamond! Free! Could you use another fabulous 100% genuine faux diamond in the Cruise household? Yes Tom Cruise, I bet you could! Just respond by . . . . .

As another example, consider a problem I faced while working on some stock-pricing software with Perl. I was getting prices that looked like "9.0500000037272". The price was obviously 9.05, but because of how a computer represents the number internally, Perl sometimes prints them this way unless print formatting is used. Normally I would just use `printf` to display with exactly two decimal digits as I did in the temperature-conversion example, but in this case I really couldn't. You see, a stock price that ends with, say, 1/8, would come out as ".125", and I wanted three digits in such cases, not two.

I boiled down my needs to "always take the first two decimal digits, and take the third only if it is not zero. Then, remove any other digits." The result is that `12.3750000000392` or the already correct `12.375` is returned as "12.375", yet `37.500` is reduced to "37.50". Just what I wanted.

So, how would we implement this? The variable `$price` contains the string in question, so let's use `$price =~ s/(\.\d\d[1-9]?)\d*/$1/`. The initial ⌈`\.`⌋ causes the match to start at the decimal point. The subsequent ⌈`\d\d`⌋ then matches the first two digits that follow. The ⌈`[1-9]?`⌋ matches an additional non-zero digit if that's what follows the first two. Anything matched so far is what we want to *keep,* so we wrap it in parentheses to capture to `$1`. We can then use `$1` in the replacement string. If this is the only thing that matches, we replace exactly what was matched with itself—not very useful. However, we go on to match other items outside the `$1` parentheses. They don't find their way to the replacement string, so the effect is that they're removed. In this case, the "to be removed" text is any extra digit, the ⌈`\d*`⌋ at the end of the regex.

Keep this example in mind, as we'll come back to it in Chapter 4 when looking at the important mechanics of just what goes on behind the scenes during a match. Some interesting lessons can be learned by playing with this example.

Automated Editing

I encountered another simple yet real-world example while working on this chapter. Connected to a machine across the Pacific, the network was particularly slow. Just getting a response from hitting RETURN took more than a minute, but I needed to make a few small changes to a file to get an important program going. In fact, all I had to do was change every occurrence of `sysread` to `read`. There were only a few such changes to make, but with the slow response, the idea of starting up a full-screen editor would have been crazy.

Here's all I did to make all the changes I needed:

```
% perl -p -i -e 's/sysread/read/g' file
```

This runs the Perl program **`s/sysread/read/g.`** (Yes, that's the whole program—the `-e` option indicates that the entire program follows on the command line.) It also uses the Perl options `-p` and `-i`, and has the program work with the file given. Briefly, that combination of options causes the substitution to be done for every line of the file, with any changes written back to the file when done.

Note that there is no explicit target string for the substitute command to work on (that is, no **`$var =~`** …) because the options I used has it implicitly work with each line of the file in turn. Also, because I used `/g`, I'm sure to replace multiple occurrences that might be in a line.

Although I applied this to only one file, I could have easily listed multiple files on the command line and Perl would have applied my substitution to each line of each file. This way, I can do mass editing across a huge set of files, all with one simple command.

A Small Mail Utility

Let's work on another example tool. Let's say we have an email message in a file, and we want to prepare a file for a reply. During the preparation, we want to quote the original message so we can easily insert our own reply to each part. We also want to remove unwanted lines from the header of the original message, as well as prepare the header of our own reply.

Let's say we have a message that looks like:

```
    From elvis Thu Feb 29 11:15 1997
    Received: from elvis@localhost by tabloid.org (6.2.12) id
KA8CMY
    Received: from tabloid.org by gateway.net.net (8.6.5/2) id
N8XBK
    Received: from gateway.net.net Thu Feb 29 11:16 1997
    To: jfriedl@ora.com (Jeffrey Friedl)
    From: elvis@tabloid.org (The King)
    Date: Thu, Feb 29 1997 11:15
    Message-Id: <1997022939939.KA8CMY@tabloid.org>
    Subject: Be seein' ya around
    Content-Type: text
    Reply-To: elvis@hh.tabloid.org
    X-Mailer: Madam Zelda's Psychic Orb [version 2.4 PL23]

    Sorry I haven't been around lately. A few years back I
checked
    into that ole heartbreak hotel in the sky,
ifyaknowwhatImean.
    The Duke says "hi".
            Elvis
```

The header has interesting fields—date, subject, and so on—but also much that we are not interested in that we'll want to clean up or remove altogether. If the script we're about to write is called *mkreply,* and the original message is in the file *king.in,* we would make the reply template with:

```
% perl -w mkreply king.in > king.out
```

(In case you've forgotten, the `-w` is option enables extra Perl warnings; ☞ 34.)

We want the resulting file, *king.out*, to contain something like:

```
To: elvis@hh.tabloid.org (The King)
From: jfriedl@ora.com (Jeffrey Friedl)
Subject: Re: Be seein' ya around

On Thu, Feb 29 1997 11:15 The King wrote:
|> Sorry I haven't been around lately. A few years back I checked
|> into that ole heartbreak hotel in the sky, ifyaknowwhatImean.
|> The Duke says "hi".
```

Let's analyze this. To print out our new header, we need to know the destination address (in this case `elvis@hh.tabloid.org`), the recipient's real name (`The King`), our own address and name, as well as the subject. Additionally, to print out the introductory line for the message body, we need to know the date.

The work can be split into three phases:

- extract information from the message header
- print out the reply header
- print out the message, indented by `'|>•'`

I'm getting a bit ahead of myself—we can't worry about processing the data until we determine how to read the data into the program. Fortunately, Perl makes this a breeze with the magic "**<>**" operator. This funny-looking construct gives you the next line of input when you assign from it to a normal `$variable`. The input comes from files listed after the Perl script on the command line (from *king.in* in the above example).

Don't confuse the two-character operator **<>** with the shell's "> *filename"* redirection or Perl's greater-than/less-than operators. It is just Perl's funny way to express a kind of a `getline()` function.

Once all the input has been read, **<>** conveniently returns an undefined value (which is interpreted as a Boolean false), so we can use the following to process a file:

```
while ($line = <>) {
... work with $line here ...
}
```

We'll use something similar for our email processing, but the nature of the task means we need to process the header specially. The header includes everything before the first blank line; the body of the message follows. To read only the header, we might use:

```
#  Process the header
while ($line = <>)
   if ($line =~ m/^\s*$/) {
       last; #  stop processing within this while loop, continue below
   }
  . . . process header line here . . .
}
. . . processing for the rest of the message follows . . .
   ⋮
```

We check for the header-ending blank line with the expression ⌈`^\s*$`⌋. It checks to see whether the target string has a beginning (as all do), followed by any number of whitespace characters (although we aren't really expecting any), followed by the end of the string.* We can't use the simple **`$line eq ""`** for reasons I'll explain later. The keyword `last` breaks out of the enclosing `while` loop, stopping the header-line processing.

So, inside the loop, after the blank-line check, we can do whatever work we like with the header lines. We will need to extract information, such as the subject and date.

To pull out the subject, we can employ a popular technique we'll use often:

```
if ($line =- m/^Subject: (.*)/) {
    $subject = $1;
}
```

This attempts to match a string beginning with `'Subject:•'`. Once that much of the regex matches, the subsequent ⌈`.*`⌋ matches whatever else is on the rest of the line. Since the ⌈`.*`⌋ is within parentheses, we can later use `$1` to access the text of the subject. In our case, we just save it to the variable `$subject`. Of course, if the regex doesn't match the string (as it won't with most), the result for the `if` is false and `$subject` isn't set for that line.

Similarly, we can look for the `Date` and `Reply-To` fields:

```
if ($line =~ m/^Date: (.*)/) {
    $date = $1;
}
if ($line =~ m/^Reply-To: (.*)/) {
    $reply_address = $1;
}
```

The `From`: line involves a bit more work. First, we want the one that begins with `'From:'`, not the more cryptic first line that begins with `'From•'`. We want:

```
From: elvis@tabloid.org (The King)
```

It has the originating address, as well as the name of the sender in parentheses; our goal is to extract the name.

To match up through the address, we can use ⌈`^From:•(\S+)`⌋. As you might guess, ⌈`\S`⌋ matches anything that's *not* whitespace (☞44), so ⌈`\S+`⌋ matches up until the first whitespace (or until the end of the target text). In this case, that's the originating address. Once that's matched, we want to match whatever is in parentheses. Of course, we also need to match the parentheses themselves, and use ⌈`\(...\)`⌋ to do that (escaping the parentheses to remove their special metacharacter meaning). Inside the parentheses we want to match anything—anything except another

---

* I use the word "string" instead of "line" because, although not an issue with this particular example, Perl can apply a regex to a string that contains a multi-line hunk of text, and the anchors caret and dollar (normally) match only at the start and end of the string as a whole (general discussion ☞81; Perl-specific details ☞232). In any case, the distinction is not vital here because, due to the nature of our algorithm, we *know* that `$line` will never have more than one logical line.

A Warning about 「.*」

The expression 「`.*`」 is often used to mean "a bunch of anything," since dot can match anything (or, in some tools, Including Perl (usually), anything except a newline and/or null) and star means that any amount is allowed but none required. This can be quite useful.

However, there are some hidden "gotchas" that can bite the user who doesn't truly understand the implications of how it works when part of a larger expression. Chapter 4 discusses this in depth.

parenthesis! That's 「`[^()]*`」. (Remember, the character-class metacharacters are different from the "normal" regex metacharacters; inside a character class, parentheses are not special and do not need to be escaped.)

So, putting this all together we get:

> 「`^From:•(\S+)•\(([^()]*)\)`」.

At first it might be a tad confusing with all those parentheses, so Figure 2-4 shows it more clearly.

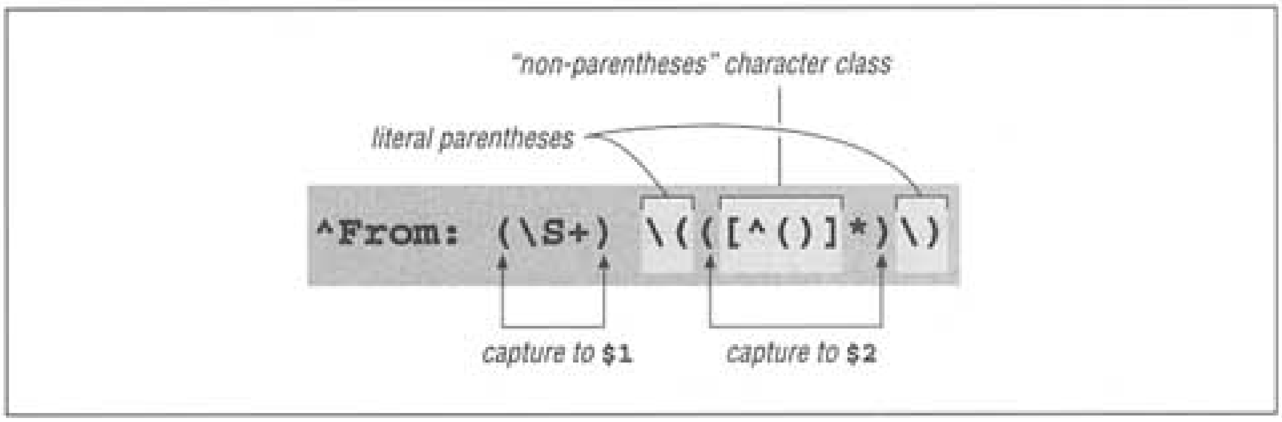


Figure 2-4:
Nested parentheses; $1 and $2

When the regex from Figure 2-4 matches, we can access the sender's name as `$2`, and also have `$1` as a possible return address:

```
if ($line =~ m/^From: (\S+) \(([^()]*)/)  {
    $reply_address = $1;
    $from_name = $2;
}
```

Since not all email messages come with a `Reply-To` header line, we use `$1` as a provisional return address. If there turns out to be a `Reply-To` field later in the header, we'll overwrite `$reply_address` at that point. Putting this all together, we end up with:

```
while ($line = <>) {
    if ($line =~ m/^\s*$/ ) { # if we have an empty line . . .
        last; # this immediately ends the 'while' loop.
    }
    if ($line =~ m/^Subject: (.*)/) {
        $subject = $1;
    }
    if ($line =~ m/^Date: (.*)/) {
        $date = $1;
    }
    if ($line =~ m/^Reply-To: (\S+)/) {
        $reply_address = $1;
    }
    if ($line -- m/^From: (\S+) \(([^()]*)\)/) {
        $replyaddress = $1;
        $from_name = $2;
    }
}
```

Each line of the header is checked against all the regular expressions, and if it matches one, some appropriate variable is set. Many header lines won't be matched by any of the regular expressions. In those cases, we end up just ignoring the line.

Once the while loop is done, we are ready to print out the reply header:

```
print "To: $reply_address ($from_name)\n";
print "From: Jeffrey Friedl <jfriedl\@ora.com>\n";
print "Subject: Re: $subject\n";
print " \n" ; # blank line to separate the header from message body.
```

Notice how we add the `Re:` to the subject to informally indicate that it is a reply. Just before printing the body of the message, we can add:

```
print "On $date $from_name wrote:\n";
```

Now, for the rest of the input (the body of the message), we want to print each line with `'|>●'` prepended:

```
while ($line = <>)  {
    print "I> $line";
}
```

Here, we don't need to provide a newline because we know that `$line` will contain one from the input.

It is interesting to see that we can rewrite the line to prepend the quoting marker, **`print "|> $ine"`**, using a regex construct:

```
$line =~ s/^/|> /;
print $line;
```

The substitute searches for ⌈`^`⌋ and (of course) immediately matches it at the beginning of the string. It doesn't actually match any characters, though, so the substitute "replaces" the "nothingness" at the beginning of the string with `'|>•'` in effect, it inserts `'|>•'` at the beginning of the string. It's a novel use of a regular expression that is gross overkill in this particular case, but we'll actually see something quite similar at the end of this chapter. (Hint: you already saw it at the start of this chapter, although I wouldn't expected you to have noticed it.)

Real-world problems, real-world solutions

It's hard to present a real-world example without pointing out its real-world shortcomings. First, as I have commented, the goal of these examples is to show regular expressions in action, and the use of Perl is simply a vehicle to do so. Again, the Perl code I've used here is not necessarily the most efficient or even the best approach, but, hopefully, it clearly shows the regular expressions at work. A "real" Perl program doing what we've just done would likely be about two lines long.*

Also, the real world is far more complex than indicated by the simple problem addressed here. A `From:` line can appear in various different formats, only one of which our program can handle. If it doesn't match our pattern exactly, the `$from_name` variable will never get set and so will be undefined (which is a kind of "no value" value) when we attempt to use it. On one hand, the solution is to update the regex to handle all the different address/name formats, but that's a bit much for this chapter. (Look for it at the end of Chapter 7.) As a first step, after checking the original message (and before printing the reply template), we can put:**

```
if (    not defined($reply_address)
     or not defined($from_name)
     or not defined($subject)
     or not defined($date)
{
     die "couldn't glean the required information!";
}
```

Perl's defined function indicates whether the variable has a value or not, while the die function issues an error message and exits the program.

Another consideration is that our program assumes that the `From:` line will appear before any `Reply-To:` line. If the `From:` line comes later, it will overwrite the `$reply_address` we took from the `Reply-To:` line.

---

* I may be exaggerating slightly, but as the first Perl program in this chapter might indicate, Perl programs can pack a lot of computational power into a short space. This is partially due to its powerful regex handling.

** In the snippet of Perl code, I use constructs not available before Perl Version 5 (it allows the example to be more readable). Users of older versions should replace `not` by `'!'`, and `or` by `'||'`.

The "real" real world

Email is produced by many different types of programs, each following their own idea of what they think the standard should be, so email can be tricky to handle. As I discovered while attempting to write some code in Pascal, it can be *extremely* difficult without regular expressions. So much so, in fact, that I found it easier to write a Perl-like regex package in Pascal rather than attempt to do everything in raw Pascal! I had taken the power and flexibility of regular expressions for granted until I ran into a world without them. It was not a pretty sight.

That Doubled-Word Thing

The double-word problem in Chapter 1 hopefully whetted your appetite for the power of regular expressions. I teased you at the start of this chapter with a cryptic bunch of symbols I called a solution. Now that you've seen a bit of Perl, you hopefully understand at least the general form the `<>`, the three `s/…/…/`, the `print`. Still, it's rather heady stuff! If this chapter has been your only exposure to Perl (and these chapters your only exposure to regular expressions), this example is probably a bit beyond what you want to be getting into at this point.

When it comes down to it, I don't think the regex is really so difficult. Before looking at the program again, it might be good to review the specification found at the start of Chapter 1, and to see a sample run:

```
  % perl -w FindDbl ch0l.txt
  ch0l.txt: check for doubled words (such as this this), a
common problem with
  ch0l.txt: * Find doubled words despite capitalization
differences, such as with 'The
  ch0l.txt: the...', as well as allow differing amounts of
whitespace (space, tabs,
  ch0l.txt: Wide Web pages, such as to make a word bold: 'it is
<B>very</B>
  ch0l.txt: very important...'
  ch0l.txt: /\<(1,000,000/million/thousand thousand)/. But
alternation can't be
  ch0l.txt: of this chapter. If you knew the the specific
doubled word to find (such
     ⋮
```

Let's look at the program now. This time, I'll use some of the nifty features of modern Perl, such as regex comments and free spacing. Other than this type of syntactic fluff, the version on the next page is identical to the one at the start of this chapter. It uses a fair number of things we haven't seen. Let me briefly explain it and some of the logic behind it, but I direct you to the Perl manpage for details (or, if regex-related, to Chapter 7). In the description that follows, "magic" means "because of a feature of Perl that you might not be familiar with yet."

1 Because the doubled-word problem must work even when the doubled words are split across lines, I can't use the normal line-by-line processing I used with the mail utility example. Setting the special variable `$/` (yes, that's a variable) as shown puts the subsequent `<>` into a magic mode such that it will return not single lines, but more-or-less paragraph-sized chunks. The

Double-word example in modern Perl

```
    $/ = ".\n";   1 #  a special "chunk-mode"'; chunks end with a period-newline combination

    while (<>)   2
    {
        next unless s   3
         {#  (regex starts here)
               ###      Need to match one word:
               \b                    #  start of word....
               ( [a-z]+ )            #  grab word, filling $1(and \1).
               ###  Now need to allow any number of spaces and/or <TAGS>
               (                   #  save what intervenes to $2.
                   (               #  ($3-parens onlyforgrouping the alternation)
                     \s              #  whitespace (includes newline, which is good).
                     |                 #  -or-
                      <[^>]+>      #  item like < TAG>.
                   )+              #  need at least one of the above, but allow more.
              )

               ###  Now match the first word again:
               (\l\b)              #  \b  ensures not embedded. This copy saved to $4.
            #  (regex ends here)
         }
         #  Above is the regex. Replacement string, below, followed by the modifiers, /i, /g, and /x
         "\e[7m$1\e[m$2\e[7m$4\e[m"igx;    4

         s/^ ([^\e]*\n)+//mg;    5      #  Remove any unmarked lines.
         s/^/$ARGV: /mg;         6      #  Ensure lines begin with filename.
         print;
     }
```

value returned will be just one string, but a string that could potentially contain many of what we would consider to be logical lines.

2 Did you notice that I don't assign the value from `<>` to anything? When used as the conditional of a `while` like this, `<>` magically assigns the string to a special default variable.* That same variable holds the default string that `s/…/…/` works on, and that `print` prints. Using these defaults makes the program less cluttered, but also less understandable to someone new to the language, so I recommend using explicit operands until you're comfortable.

3 The `s` on this line is the one from the substitute operator, `s/…/…/`, which is much more flexible than we've seen so far. A wonderful feature is that you don't have to use slashes to delimit the regex and replacement string; you can use other symbols such as the `s{regex}"replacement"` I use here. You have to look all the way down to 4 to see that I use the `/x` modifier with this substitution. `/x` allows you to use comments and free spacing within the regex (but not the replacement string). Those two dozen or so lines of regex are mostly comments the "real" regex itself is byte-for-byte identical to the one at the start of this chapter.

---

* The default variable is `$_` (yes, that's a variable too). It's used as the default operand for many functions and operators.

The **`next unless`** before the substitute command has Perl abort processing on the current string (to continue with the next) if the substitution didn't actually do anything. There's no need to continue working on a string in which no doubled-words are found.

4 The replacement string is really just "`$1$2$4`" with a bunch of intervening ANSI escape sequences that provide highlighting to the two doubled words, but not to whatever separates them. These escape sequences are `\e[7m` to begin highlighting, and `\e[m` to end it. (`\e` is Perl's regex and string shorthand for the ASCII escape character, which begins these ANSI escape sequences.)

Looking at how the parentheses in the regex are laid out, you'll realize that "`$1$2$4`" represents exactly what was matched in the first place. So other than adding in the escape sequences, this whole substitute command is essentially a (slow) no-op.

We know that `$1` and `$4` represent matches of the same word (the whole point of the program!), so I could probably get by with using just one or the other in the replacement. However, since they might differ in capitalization, I use both variables explicitly.

5 Once the substitution has marked all doubled words in the (possibly many-logical-lined) string, we want to remove those logical lines which don't have an escape character (leaving only the lines of interest in the string).* The `/m` used with this substitution and the next makes the substitution magically treat the target string (the `$_` default mentioned earlier) as logical lines. This changes the meaning of caret from start-of-*string* to "*start-of-logical-line*", allowing it to match somewhere in the middle of the string if that somewhere is the start of a logical line. The regex ⌈`^([^\e]*\n)+`⌋ finds sequences of non-escapes that end in a newline. Use of this regex in the substitute causes those sequences to be removed. The result is that only logical lines that have an escape remain, which means that only logical lines that have doubled words in them remain.

6 The variable `$ARGV` magically provides the name of the input file. Combined with `/m` and `/g`, this substitution tacks the input filename to the beginning of each logical line remaining in the string. Cool!

Finally, the `print` spits out what's left of the string, escapes and all. The `while` loop repeats the same processing for all the strings (paragraph-sized chunks of text) that are read from the input.

---

* This logic assumes that the input file doesn't have any ASCII escape characters of its own. If it did, this program could report lines in error.

There's nothing special about Perl . . .

As I emphasized early in this chapter, Perl is a tool used to show the concepts. It happens to be a very useful tool, but I again want to stress that this problem can be solved just as easily in other languages. Chapter 3 shows a similar type of solution with GNU Emacs. As a direct comparison, the listing below shows a solution written in the Python language. Even if you've never seen Python before, you can still get a feel for the different way it handles regular expressions.

Double-word example in Python

```
import sys; import regex; import regsub

## Prepare the three regexes we'll use
regl = regex.compile(
        '\\b\([a-z]+\)\(\([\n\r\t\f\v ]\|<[^>]+>\)+\)\(\\1\\b\)',
        regex.casefold)
reg2 = regex.compile('^\([^\033]*\n\)+')
reg3 = regex.compile('^\(.\)')

for filename in sys.argv[l:]:                # for each file. . . . .
    try:
        file = open(filename)                # try opening file
    except IOError, info:
        print '%s: %s' % (filename, info[l]) # report error if couldn't
        continue                             # and also abort this iteration

    data = file.read()    ## Slurp whole file to 'data, apply regexes, and print
    data = regsub.gsub(regl, '\033[7m\\1\033[m\\2\033[7m\\4\033[m', data)
    data = regsub.gsub(reg2, '', data)
    data = regsub.gsub(reg3, filename + ': \\1', data)
    print data,
```

Most of the changes are in the mechanics of shoving the bits around. Perl has been designed for text processing and magically does many things for you. Python is much more "clean" in that its interfaces are extremely consistent, but this means that much of the day-to-day work of opening files and such must be done manually.[*]

Python's regex flavor is opposite from Perl's and *egrep's* in one respect: it requires a bazillion backslashes. For example, ⌈`(`⌋ is not a metacharacter—⌈`\(…\)`⌋ provides grouping and capturing. It also doesn't support `\e` as a shorthand for the escape character, so I insert it directly by its ASCII `\033` encoding. Other than one detail about ⌈`^`⌋ that I'll explain in a moment, these differences are superficial; the regular expressions are functionally identical to the ones used in the Perl example.[**]

---

[*] This is exactly what Perl lovers love about Perl and hate about Python, and what Python lovers love about Python and hate about Perl.

** Okay, I admit that there's one other difference that *might* present itself. Perl's ⌈`\s`⌋, replaced here by ⌈`[\n\r\t\f\v•]`⌋, trusts the `C` library that it was compiled with as far as deciding what is and isn't whitespace. My Python regex lists explicitly what I consider whitespace.

Another interesting thing is that the replacement string for `gsub` ("global substitution," analogous to Perl's `s/…/…/)` uses the same `\\1` that is used in the regex itself. To access the same information outside of both the regex and the replacement, Python provides a third notation, `regex.group(1)`. You'll remember that Perl uses ⌈`\1`⌋ within the regex, `$1` everywhere else. Similar concepts, different approaches.

One important non-superficial, regex-related difference is that Python's ⌈`^`⌋ considers a final newline to be the start of an empty line. This manifests itself with an extra `filename:` line if the third `regex` is the same ⌈Ù⌋ used with Perl. To get around this difference, I simply required the third regex to match something after the caret, replacing that something with itself. The effect is that it won't match after the end of the last logical line.

3

Overview of Regular Expression Features and Flavors

In this chapter:

• A Casual Stroll Across the Regex Landscape
• At a Glance
• Care and Handling of Regular Expressions
• Engines and Chrome Finish
• Common Metacharacters
• Guide to the Advanced Chapters

Now that we have a feel for regular expressions and two diverse tools that use them (*egrep* and Perl), you might think we're ready to dive into using them wherever they're found. As I hope the brief look at Python at the end of the previous chapter indicated, the way regular expressions appear, are used, and act can vary *wildly* from tool to tool. It is not as bad as it sounds because you quickly get used to tools you use often, but you should be aware of the differences out there.

The Goal of This Chapter

Let me start by saying that it is *not* the goal of this chapter to provide a reference for any particular tool's regex features, nor to teach how to use regexes in any of the various tools and languages mentioned as examples. It is also not a goal to provide an exhaustive list of metacharacters found throughout the world today.

It *is* a goal to provide a global perspective on regular expressions and the tools that implement them. If you lived in a cave using only one particular tool, you could live your life without caring about how other tools (or other versions of the same tool) might act differently. Since that's not the case, knowing something about your utility's computational pedigree adds interesting and valuable insight.

Acquainting yourself with how different tools address the same concerns is beneficial, and this chapter provides a small taste—the main meal is found in later chapters. Cognizance of the various issues helps you acclimate to new tools more quickly: knowing what's out there can help you select the proper tool for the job.

It didn't make sense to talk about this earlier, before we had a tangible grasp of just what regexes actually were. Yet, I don't want to delay further because the added insight and global perspective will be valuable when examining the nitty-gritty details starting Chapter 4. (The "Guide to the Advanced Chapters" on page 85 is your road map to the detailed and heady remainder of the book.)

A Casual Stroll Across the Regex Landscape

I'd like to tell a short story about the evolution of some regular expression flavors and their associated programs. So grab a hot cup (or frosty mug) of your favorite brewed beverage and relax as we look at the sometimes wacky history behind the regular expressions we have today. Again, the idea is to add color to our regex understanding and to develop a feeling as to why "the way things are" are the way things are.

The World According to Grep

I'll start our story with an ancestor of *egrep* called *grep,* arguably the most common regex-related program today. It has been available on Unix systems since the mid-1970s and has been ported to virtually every modern system. There are dozens of different (some wildly so) versions of *grep* for DOS.

You might think that such a common, veteran program would be standardized, but sadly, that's not the case. Put yourself in "reading for enjoyment" mode as we go back to the beginning and follow the trail of how things have developed.

Pre-grep history

The seeds of regular expressions were planted in the early 1940s by two neurophysiologists, Warren McCulloch and Walter Pitts, who developed models of how they believed the nervous system worked at the neuron level.[*] Regular expressions became a reality several years later when mathematician Stephen Kleene formally described these models in an algebra he called *regular sets.* He devised a simple notation to express these regular sets, and called them *regular expressions.*

Through the 1950s and 1960s, regular expressions enjoyed a rich study in theoretical mathematics circles. Robert Constable has written a good summary[**] for the mathematically inclined. Although there is evidence of earlier work, the first published computational use of regular expressions I have actually been able to find is Ken Thompson's 1968 article *Regular Expression Search Algorithm*[***] in which he

---

* "A logical calculus of the ideas immanent in nervous activity," first published in *Bulletin of Math. Biophysics* 5 (1943) and later reprinted in *Embodiments of Mind* (MIT Press, 1965). The article begins with an interesting summary of how neurons behave (did you know that intra-neuron impulse speeds can range from 1 all the way to 150 meters per second?), and then descends into a pit of formulae that is, literally, all Greek to me.

** Robert L. Constable, "The Role of Finite Automata in the Development of Modern Computing Theory," in *The Kleene Symposium,* eds. Barwise, Keisler, and Kunen (North-Holland Publishing Company, 1980), 61-83.

*** *Communications of the ACM,* Vol.11, No 6, June 1968.

describes a regular-expression compiler that produced `IBM 7094` object code. This led to his work on *qed,* an editor that formed the basis for the Unix editor *ed.* The regular expressions of *ed* were not as advanced as those in *qed,* but they were the first to gain widespread use in non-technical fields. *ed* had a command to display lines of the edited file which matched a given regular expression. The command, " `g/`*Regular Expression*`/p` ", was read "Global Regular Expression Print." This particular function was so useful that it was made into its own utility, and *grep* was born.

Grep's metacharacters

The regular expressions supported by these early tools were quite limited when compared to *egrep's.* The metacharacter `*` was supported, but `+` and `?` were not (the latter's absence being a particularly strong drawback). *grep's* grouping metacharacters were `\(…\)`, with unescaped parentheses representing literal text.* *grep* supported line anchors, but in a limited way. If `^` appeared at the beginning of the regex, it was a metacharacter matching the beginning of the line, just as it is with *egrep* and Perl. Otherwise, it wasn't a metacharacter at all and just matched a literal circumflex. Similarly, `$` was the end-of-line metacharacter only at the end of the regex. The upshot was that you couldn't do something like ⌈`end$|^start`⌋. But that's okay, since alternation wasn't supported either!

The way metacharacters interact is also important. For example, perhaps *grep's* largest shortcoming was that star could not be applied to a parenthesized expression, but only to a literal character, a character class, or dot. So, in *grep,* parentheses were useful only for remembering matched text (such as with ⌈`\([a-z]+\)•\1`⌋ to match repeated words), but not for general grouping. In fact, some early versions of *grep* didn't even allow nested parentheses.

The Times They Are a Changin'

Grep evolves

Although many systems have *grep* today, you'll note that I've been using past tense. The past tense refers to the flavor of the old versions, now upwards of 20 years old. Over time, as technology advances, older programs are sometimes retrofitted with additional features and *grep* has been no exception.

Along the way, AT&T Bell Labs added some new features, such as incorporating the `\{`*`min,max`*`\}` notation (mentioned in Chapter 1) from the program *lex*. They also fixed the `-y` option, which in early versions was supposed to allow case-insensitive matches but worked only sporadically. Around the same time, people at Berkeley added start- and end-of-word metacharacters and renamed `-y` to `-i`. Unfortunately, you still couldn't apply star or the other quantifiers to a parenthesized expression.

---

* Historical trivia: *ed* (and hence *grep*) used escaped parentheses rather than unadorned parentheses as delimiters because Ken Thompson felt regular expressions would be used to work primarily with C code, where needing to match raw parentheses would be more common than using backreferences.

Egrep evolves

By this time, Alfred Aho had written *egrep,* which provided most of the richer set of metacharacters described in Chapter 1. More importantly, he implemented them in a completely different (and generally better) way. (Implementation methods and what they mean to us as users are the focus of the next two chapters.) Not only were plus and question mark added, but they and the other the quantifiers could be applied to parenthesized expressions, *greatly* increasing the power of *egrep* regular expressions.

Alternation was added as well, and the line anchors were upgraded to "first-class" status so that you could use them most anywhere in your regex. However, *egrep* had problems as well—sometimes it would find a match but not display the result, and it didn't have some useful features that are now popular. Nevertheless, it was a vastly more useful tool.

Other species evolve

At the same time, other programs such as *awk, lex,* and *sed,* were growing and changing at their own pace. Often, developers that liked a feature from one program tried to add it to another. Sometimes, the result wasn't pretty. For example, if you want to add support for plus to *grep,* you can't just use `+`, as *grep* has a long history of '`+`' not being a metacharacter, and suddenly making it one would surprise users. Since "`\+`" is probably not something a *grep* user would otherwise normally want to type, it can safely be subsumed as the "one or more" metacharacter.

Sometimes, new bugs are introduced as features are added. Other times, an added feature is later removed. There was little to no documentation for the many subtle points that round out a tool's flavor, so new tools either made up their own style, or attempted to mimic "what happened to work" with other tools.

Multiply this by the passage of time and numerous programs and the result is general confusion (particularly when you try to deal with everything at once).* The dust settled a bit in 1986 when Henry Spencer released a regex package written in C that anyone could freely incorporate into their own program (a first at the time). Every program that used his package (there were, and are, many) provided the same consistent regex flavor (unless the author went to the trouble to change it).

---

* Such as when writing a book about regular expressions—ask me, I know!

At a Glance

Charting just a few aspects of some common tools gives a good clue to how much different things have become. Table 3-1 provides a very superficial look at the generic flavor of a few tools (it is an abridged version of Table 6-1 on page 182).

Table 3-1: A (Very) Superficial Look at the Flavor of a Few Common Tools

Feature	Modern *grep*	Modern *egrep*	awk	GNU Emacs Version 19	Perl	Tcl	vi
`*, ^, $, […]`	√	√	√	√	√	√	√
`? + |`	`\? \+ \\|`	`? + |`	`? + |`	`? + \\|`	`? + |`	`? + |`	`\? \+` ■
`grouping`	\\(…\\)	(…)	(…)	`\(…\)`	`(…)`	`(` ■ `¼)`	`\(…\)`
`word boundary`	•	`\< \>`	•	`\< \> \b, \B`	`\b, \B`	•	`\< \>`
`\w, \W`	•	√	•	√	√	•	•
`backreferences`	√	•	•	√	√	•	√

This kind of chart is often found in other books to show the differences among tools. But this chart is only the tip of the iceberg—for every feature shown, there are a dozen important issues that are overlooked, including:

- Are star and friends allowed to quantify something wrapped in parentheses?
- Does dot match a newline? Do negated character classes? What about the null character?
- Are the line anchors really *line* anchors, or just *target-string* anchors? Both? Neither? Are they first-class metacharacters, or are they valid only in certain parts of the regex?
- Are escapes recognized in character classes? What else is or isn't allowed within character classes?
- Are parentheses allowed to be nested? If so, how deeply (and how many parentheses are even allowed in the first place)?
- Are octal escapes allowed? If so, how do they reconcile the conflict with backreferences? What about hexadecimal escapes? Is it *really* the regex engine that supports octal and hexadecimal escapes, or is it some other part of the utility?
- Does ⌈`\w`⌋ match only alphanumerics, or additional characters as well? (All three programs supporting \w shown in Table 3-1 treat it differently!)
- If `\n` is supported, what exactly does it mean? Are there other convenience metacharacters?

Many issues must be kept in mind, even with a tidy little summary like Table 3-1 as a superficial guide. If you realize that there's a lot of dirty laundry behind that nice facade, it's not difficult to keep your wits about you and deal with it. Bear in mind that many programs have differences from vendor to vendor and from version to version. Since Table 3-1 is just an overview, I've stuck with the commonly found features of recent versions. The GNU versions of many common tools, for example, are almost always more powerful and robust than other versions.

The differences in the semantics of how a match is attempted (or, at least, how it appears to be attempted) is an *extremely important* issue often overlooked in other overviews. Once you understand that something such as ⌈`(Jul|July)`⌋ in *awk* needs to be written as ⌈`\(Jul\|July\)`⌋ for GNU Emacs, you might think that everything will then be the same from there. That's not always the case—there are certainly situations where seemingly comparable expressions will end up matching differently (such as these two). We dive into this important issue in the next chapter.

Of course, what a tool can *do* with a regular expression is often more important than the flavor of its regular expressions. For example, even if Perl's expressions were less powerful than *egrep's,* Perl's flexible use of regexes provides for more usefulness overall. So even though Table 3-1 provides an interesting look at some combinations of flavors, it shouldn't be used to "judge" a tool's usefulness. I delve more deeply into the subject of examining a tool's regex flavor in Chapter 6.

POSIX

POSIX, short for Portable Operating System Interface, is a standard for ensuring portability across operating systems. Within this ambitious standard are specifications for regular expressions and many of the traditional tools that use them.

In an effort to reorganize the mess that Table 3-1 hints at, POSIX distills the various common flavors into just two classes of regex flavor, *Basic Regular Expressions* (BREs), and *Extended Regular Expressions* (EREs). Fully POSIX-compliant tools use one of the flavors, or one with a few select tool-specific enhancements. Table 3-2 summarizes the metacharacters of the two flavors.

Table 3-2: Overview of POSIX Regex Flavors

Regex Feature	**BREs**	**EREs**
`dot, ^, $, […], [^…]`	√	√
`*,+,?,` {*min, max*}	`*,·,·,` `\`*min, max*`\}`	`*, +, ?,` {*min, max*}
`grouping`	`\(… \)`	`(…)`
`can apply quantifiers to parentheses`	√	√
`backreferences`	\1 through \9	•
`alternation`	•	√

Table 3-2, like Table 3-1, is quite superficial. For example, BRE's dollar anchor is valid only at the end of the regex (and optionally, at the discretion of each implementation, before a closing parenthesis as well). Yet in EREs, dollar is valid everywhere except within a character class. We'll see more details throughout this chapter.

I admit some ignorance with many POSIX items, as I've never used a tool that supported them fully. However, many popular tools selectively embrace some aspects of the POSIX standard, and so even for those on a non-POSIX platform, it pays to be aware. Let's start by looking at the POSIX *locale.*

POSIX locale

One feature of the POSIX standard is the notion of a *locale,* settings which describe language and cultural conventions such as the display of dates, times, and monetary values, the interpretation of characters in the active encoding, and so on. Locales aim at allowing programs to be internationalized. It is not a regex-specific concept, although it can affect regular-expression use.

For example, when working with a locale that describes the *Latin-1* `(ISO-8859-1)` encoding, and À are considered "letters" (encoded outside the ASCII range, most tools would otherwise consider them simply as raw data), and any application of a regex that ignores capitalization would know to treat them as identical.

Another example is ⌈`\w`⌋, commonly provided as a shorthand for a "word-constituent character" (ostensibly, the same as ⌈`[a-zA-Z0-9]`⌋). This feature is not required by POSIX, but it is allowed. If supported, ⌈`\w`⌋ would know to allow all letters and digits defined in the locale, not just those in the English alphabet.

POSIX collating sequences

A locale can define named *collating sequences* to describe how to treat certain characters, or sets of characters, for sorting and such. For example, the Spanish `ll` (as in *tortilla*) traditionally sorts as if it were one logical character between `l` and `m`, and the German ⌋ is a character that falls between `s` and `t`, but sorts as if it were the two characters `ss`. These rules might be manifested in collating sequences named, for example, `span-ll` and `eszet`.

As with the `span-ll` example, a collating sequence can define multi-character sequences that should be taken as a *single* character as far as character classes (what POSIX calls "bracket expressions"; ☞ 79) are concerned. This means that the class in ⌈`torti[a-z]a`⌋ matches the two-character "single character" in `tortilla`. And since ⌋ is defined as being a character between s and t, the character class ⌈`[a-z]`⌋ includes it.

Backdoor support for locales

Locales can influence many tools that do not aspire to POSIX compliance, sometimes without their knowledge! Most utilities are written in C or C++ and often use standard C library functions to determine which bytes are letters, digits, and the like. If the non-POSIX utility is compiled on a system with a POSIX-compliant C library, some support can be bestowed, although the exact amount can be hit or miss. For example, the tool's author might have used the C library functions for capitalization issues, but not for `\w` support.*

Some utilities explicitly try to embrace this support in their regular expressions, but often to only a small extent. Perl, Tcl, and GNU Emacs are examples. Perl's `\w` and case-insensitive matches honor a locale, as described above, but its dot and character-class ranges do not. We'll see more examples as we look at individual metacharacters, starting in "Common Metacharacters" (☞ 71).

Care and Handling of Regular Expressions

Closely associated with regular expressions is the syntactic packaging that tells an application "hey, here's a regex, and this is what I want you to do with it." *egrep is* a simple example because the regular expression is expected as an argument on the command line   if you need anything extra, such as the singlequotes I used throughout the first chapter, it is only to satisfy the command shell, not *egrep.* In more complex systems, where regular expressions are used in various situations, more-complex packaging is required to inform the system exactly what the regex is, and how it should be used.

This section is a whirlwind tour through some of the ways that programs wield their regexes. I'll start with Perl again, but I'll comment on other tools. I won't go into much depth, the goal being to learn a few key concepts to get a feel for the different ways things are done.

Identifying a Regex

In the previous chapter, we looked at Perl, a full-featured language that allows many kinds of expressions. With all the expression types, you need to tell Perl when you are actually using a regular-expression expression. Use `m/…/` around your regex to indicate a regex search, and use `=~` to link it to the text to be searched. (Actually, if you like, you can drop the `m` or, if you like, you can use any other symbol instead of the slashes   nifty!) Remember, that the slashes are not part of the regex itself, but are merely the delimiters showing where the regex is in the script. It's the syntactic packaging I mentioned before.

---

* For example, the URL encoder on page 257, which now uses ⌈`[^a-zA-Z0-9]`⌋, used ⌈`\W`⌋ in earlier printings of this book until a friend ran into a problem in which his version of Perl treated certain non-ASCII bytes as "●", "é", and the like. He expected these bytes to be ⌈`\W`⌋. but his Perl considered them ⌈`\w`⌋, causing unexpected results.

Doing Something with the Matched Text

Of course, a regular expression is good for more than just finding text. The Perl substitute command **`$var =~ s`***/regex/replacement/* we saw in Chapter 2 is a good example. It searches the string within the variable for text that can be matched by the given regular expression, and replaces that text with the given replacement string. Appending `/g` to the command replaces the text "globally" within the string. This means that after the first replacement, the rest of the string is checked for additional replacements.

Remember that with a search-and-replace command, the replacement string is *not* a regular expression. Still, like many constructs, it has its own metacharacters. Consider:

```
$var =~ s/[0-9]+/<CODE>$&<\/CODE>/g
```

This command effectively wraps each number in `$var` with `<CODE>`…`</CODE>`. The replacement string is `<CODE>$&<\/CODE>`. The backslash escapes the command-delimiting slash to allow it to appear in the replacement string. The `$&` is a Perl variable that represents the text that was matched by the last regular-expression match (the ⌈`[0-9]+`⌋ in the first part of the command).

You can even use a different command delimiter. If we use an exclamation point instead of the slash we normally use, the last example appears as:

```
$var =~ s![0-9]+!<CODE>$&</CODE>!g
```

Notice that there is no need to escape the replacement string's `/`, as it is now not the command delimiter and is not otherwise special.

The important thing to remember about things like

```
$var =~ m/[0-9]+/;
$var =~ s/[0-9]+/a number/g;
$var =~ s![0-9]+!<CODE>$&</CODE>!g;
```

is that in all cases, the regular expression is the same. What might differ is what the tool, Perl, does with it.

Other Examples

As can be expected, not every program provides the same features or uses the same notations to express the same concepts. Let's look at a few other tools. (Most are examined in more detail in Chapter 6.)

Awk

awk uses /*regex*/ to perform a match on the current input line, and uses `var` ~… to perform a match on other data. You can see where Perl got its notational influence in this respect. (Perl's substitution operator, however, is modeled after *sed's.)*

The early versions of *awk* didn't support a regex substitution, but modern versions have the `sub(…)` function. Something such as **`sub(/mizpel/, "misspell")`** applies the regex ⌜`mizpel`⌟ to the current line, replacing the first match with `misspell`. Note how this compares to Perl's `s/mizpel/misspell/`.

To replace all matches within the line, *awk* does not use any kind of `/g` modifier, but a different function altogether: **`gsub(/mizpel/, "misspell")`**

Tcl

Tcl takes a different approach which might look confusing if you're not familiar with Tcl's quoting conventions (but that's okay, because theses examples just give you a feel for how regexes can be handled differently). To correct our misspellings with Tcl, we might use:

```
regsub mizpel $var misspell newvar
```

This checks the string in the variable `var`, and replaces the first match of ⌜`mizpel`⌟ with `misspell`, putting the resulting text into the variable `newvar`. Neither the regular expression nor the replacement string requires slashes or any other delimiter other than the separating spaces. Tcl expects the regular expression first, the target string to look at second, the replacement string third, and the name of the target variable fourth. (As with any Tcl argument, if the regex or replacement string have spaces or the like, you can delimit it with quotes.) Tcl also allows certain options to its `regsub`. For example, to replace all occurrences of the match instead of just the first, add `-all:`

```
regsub -all mizpel $var misspell newvar
```

Also, the `-nocase` option causes the regex engine to ignore the difference between uppercase and lowercase characters (just like *egrep's* `-i` flag, or Perl's `/i` modifier).

GNU Emacs

The massively powerful text editor GNU Emacs (just "Emacs" from here on in) supports *elisp* (Emacs lisp) as a built-in programming language, and provides numerous functions that deal with regular expressions. One of the main ones is `re-search-forward`, which accepts a normal string as an argument and interprets it as a regular expression. It then starts searching the text from the "current position," stopping at the first match or aborting if no match is found. (This is the function that is invoked when one invokes a search while using the editor.) For example, (`re-search-forward "main"`) searches for ⌜`main`⌟, starting at the current location in the text being edited.

As Table 3-1 shows, Emacs's flavor of regular expressions is heavily laden with backslashes. For example, ⌜`\<\([a-z]+\)\([\n•\t]\|<[^>]+>\)+\l\>`⌟ is an expression for finding doubled words (similar to the problem in the first chapter). We couldn't use this directly, however, because the Emacs regex engine doesn't understand `\t` and `\n`. Emacs doublequoted strings, however, do. Unlike Perl and *awk* (but like Tcl and Python), regular expressions in Emacs *elisp* scripts are often provided to the regex engine as string literals, so we can feel free to use `\t` and the like. It poses a slight problem, however, because backslashes are special to an *elisp* doublequoted string.

With *egrep,* we generally wrap the regular expression in singlequotes so that characters such as `*` and `\`, which are shell metacharacters, are safe to use in our expression. With Perl, the `m`/*regex*/ or `s`/*regex*/*replacement*/ commands accept a regular expression directly, so there is no metacharacter conflict (except, of course, with the regex delimiter itself, usually a slash). With *elisp,* there is no such easy way out. Because a backslash is a string metacharacter, you need to escape it (use `\\`) for each `\` that you actually want in the regular expression. Combined with *elisp's* propensity for backslashes, the result generally looks like a row of scattered toothpicks. Here's a small function for finding the next doubled word:

```
    (defun FindNextDbl ()
      "move to next doubled word, ignoring <…> tags"
(interactive)
      (re-search-forward "\\<\\([a-z]+\\)\\([\n
\t]\\|<[^>]+>\\)+\\1\\>")
    )
```

Combine that with **`(define-key global-map "\C-x\C-d" 'FindNextDbl)`** and you can use the "Control-`x` Control-`d`" sequence to quickly search for doubled words.

Python

Python is an ambitious object-oriented scripting language quite different from anything we've seen so far. Its regex flavor closely mimics that of Emacs. Well, usually—in Python, you can actually change parts of the regex flavor on the fly! Sick of all those backslashes in the Emacsesque flavor? You can get rid of them with:

```
    regex.set_syntax( RE_NO_BK_PARENS | RE_NO_BK_VBAR
```

The two items indicate that you prefer to have your expressions interpreted as using unadorned parentheses for grouping, and unadorned bar for alternation. You can guess my preference!

Python is object-oriented, including its regular expressions. You can create a "regex object" and later apply it to strings for matching or substitutions. In the snippet below, I use mixed capitalization for variable names to help distinguish them from library components. I also use the default regex flavor:

```
    MyRegex = regex.compile("\([0-9]+\)");
      ⋮
    MyChangedData = regsub.gsub(MyRegex, "<CODE>\\1</CODE>",
MyData)
```

You can guess that the string with the `<CODE>` parts is the replacement text. Within the regex itself, Python, Perl, Tcl, and Emacs all use the ⌈`\1`⌋ notation for backreferences, but unlike Perl and its `$1`, the others use that same ⌈`\1`⌋ notation within the replacement text as well.

This might raise the question of what is used elsewhere in the program, after the substitution has completed. (Perl's `$1` continues to be available as a mostly-normal variable that can be referenced as you like.) Object-oriented Python has the regex object (`MyRegex` in the example) hold the information about its last match. Perl's `$1` is Python's `MyRegex.group(1)`. (Tcl and Emacs, by the way, have different approaches altogether; ☞ 191, 196).

Never one to be boring, Python has an interesting approach for allowing a case-insensitive match: you can provide a description indicating how each and every byte (i.e., character) should be considered for comparison purposes. A description indicating that uppercase and lowercase are to be considered the same would provide for traditional case-insensitive matching, but the possibilities are wide open. Working with the *Latin-1* encoding popular on the Web (which is chock full of letters with funny growths), you specify that a "case insensitive" match ignore the funny growths. Additionally, you could have (if you wanted) ¿ match as a question mark, ¡ as an exclamation mark, and perhaps even have ¢, ¤, £ and ¥ all match equally with `$`. Essentially, it's a way of providing a set of character classes that should be applied at all levels of the match. Fantastic!

Care and Handling: Summary

As you can see, there's a wide range of functionalities and mechanics for achieving them. If you are new to these languages, it might be quite confusing at this point. But never fear! When trying to learn any one particular tool, it is a simple matter to learn its mechanisms.

One difficulty I have in showing later examples is that regular expressions aren't used in a vacuum, but with a host utility, and they are often linked with some non-regex functionality of the host tool. To make a general point, I still have to choose how to show the regex. I'll generally stick to an *egrep-awk-Perl* style flavor that is not cluttered with backslashes—converting to your favorite flavor is simple.

Engines and Chrome Finish

There's a huge difference between how a car (or in my case, a motorcycle) looks and its engine. The neighbors will compliment you on the shine and finish, but the mechanics and others in-the-know will talk about the engine. Is it an inline-4? V-8? Diesel? Dual clutched? High-performance cams? Tuned pipes? Or maybe just a few squirrels running on a treadmill? On the racetrack, all these issues play a part in every decision the driver makes. You might not think it's important if you take only short trips to the grocery store, but eventually you'll have to decide what kind of gas to get. Ask anyone stuck in the middle of the desert with a broken thingamajiggy if the shine and finish are more important, and you can guess the answer. And here's a bit of insight: if your term for something is "thingamajiggy," your chances of being able to fix it yourself are probably pretty slim.

When it comes to regular expressions, there are two distinct components to the flavor a tool provides. One component is discussed for the rest of this chapter; the other is discussed in the next.

Chrome and Appearances

The most apparent differences among regex flavors are in which metacharacters are supported. As we've seen, Perl provides some metacharacters not found in *egrep*. Note, however, that while both allow you to match word boundaries, they use different metacharacters and approaches to do so. Even more interesting, their exact idea of what constitutes a word boundary is different. As most of this chapter shows, these subtle "shine and finish" differences abound.

Engines and Drivers

The implementation of a metacharacter's meaning and the semantics of how they combine to make larger expressions are *extremely* important, even though they aren't as obvious as the differences in which metacharacters are supported. The differences in regex engine implementation is often reflected in:

- what exactly will be matched
- the speed of the match
- the information made available after the match (such as Perl's `$1` and friends)

If you wish to confidently write anything beyond trivial expressions, understanding these points, and why, is important. This is the focus of the next chapter.

Common Metacharacters

This overview of current regex metacharacters covers common items and concepts. Of course, it doesn't discuss every single one, and no one tool includes everything presented here. In one respect, this is just a summary of much of what we've seen in the first two chapters, but in light of the wider, more complex world presented at the beginning of this chapter. If this is the first time through this section, a light glance should allow you to continue on to the next chapters. You can come back here to pick up the details as you need them.

Some tools add a lot of new and rich functionality (particularly Perl) and some gratuitously change standard notations to suit their whim (such as just about anything from Microsoft). Some try to be standard, but leave the straight and narrow to fill their special needs. Although I'll sometimes comment about specific utilities, I won't address too many tool-specific concerns here. (Chapter 6 looks at *awk*, Emacs, and Tcl in more detail, and Chapter 7 at Perl.) In this section, I'll just try to cover some common metacharacters and their uses, and some concerns to be aware of. I encourage you to follow along with the manual of your favorite utility.

Character Shorthands

Many utilities provide metacharacters to represent machine-dependent control characters that would otherwise be difficult to input or to visualize:

`\a` **Alert** (e.g. to sound the bell when "printed") Usually maps to the ASCII `<BEL>` character, `007` octal.

`\b` **Backspace** Usually maps to the ASCII `<BS>` character, `010` octal. (Note ⌊`\b`⌋ often is a word-boundary metacharacter instead, as we'll see later.)

`\e` **Escape character** Usually maps to the ASCII `<ESC>` character, `033` octal.

`\f` **Form feed** Usually maps to the ASCII `<FF>` character, `014` octal.

`\n` **Newline** On most platforms (including Unix and DOS/Windows), usually maps to the ASCII `<LF>` character, `012` octal. On MacOS systems, usually maps to the ASCII `<CR>` character, `015` octal.

`\r` **Carriage return** Usually maps to the ASCII `<CR>` character. On MacOS systems, usually maps to the ASCII `<LF>` character.

`\t` **Normal (horizontal) tab** Usually maps to the ASCII `<HT>` character, `011` octal.

`\v` **Vertical tab**[*] Usually maps to the ASCII `<VT>` character, `013` octal.

Table 3-3 lists a few common tools and some of the control shorthands they provide, as well as a few things we'll see later in this chapter.

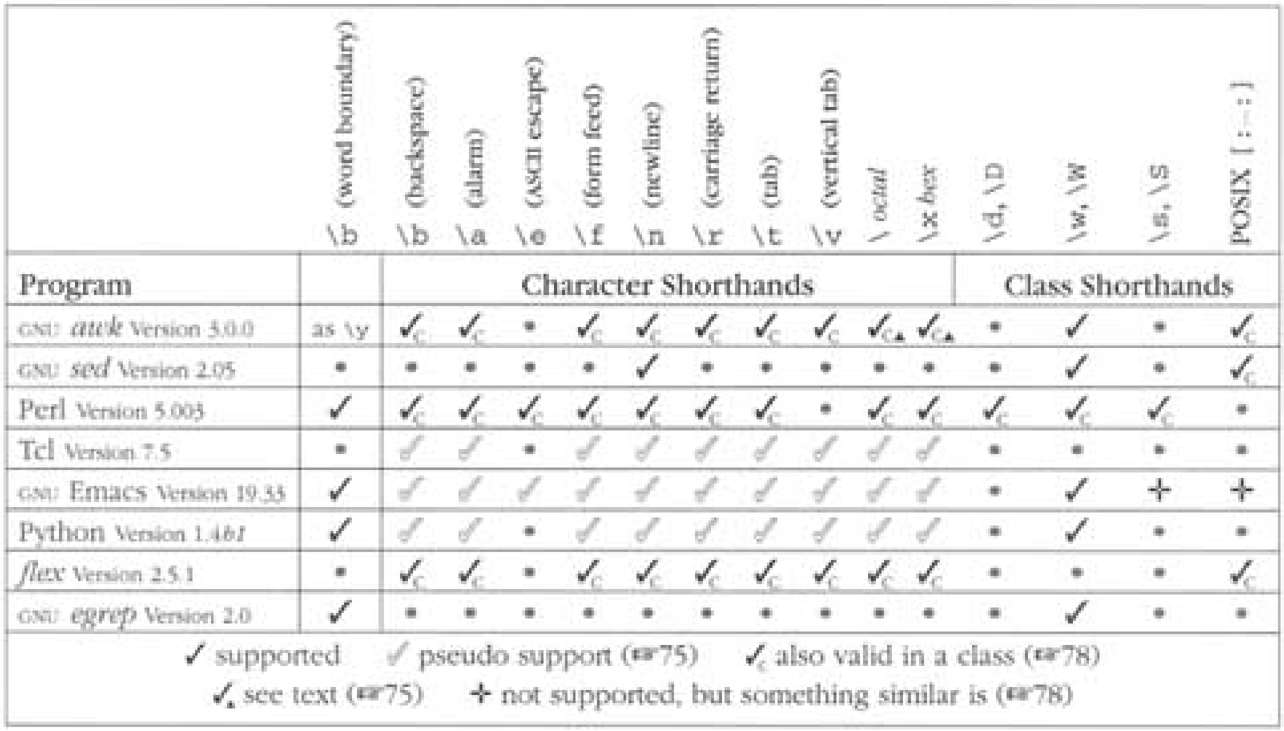
These are machine dependent?

With most tools (including every tool mentioned in this book whose source I have been able to inspect, which is most of them), many control-character shorthands are *machine dependent,* or, more pedantically, *compiler dependent.* All the tools mentioned in this book (whose source I have seen) are written in C or C++.

---

[*] A vertical tab is a character in ASCII whose relevance has fallen away with, I suppose, the demise of teletypes.

Table 3-3: A Few Utilities and Some of the Shorthand Metacharacters They Provide



	`\b` (word boundary)	`\b` (backspace)	`\a` (alarm)	`\e` (ASCII escape)	`\f` (form feed)	`\n` (newline)	`\r` (carriage return)	`\t` (tab)	`\v` (vertical tab)	`\`*octal*	`\x`*hex*	`\d`, `\D`	`\w`, `\W`	`\s`, `\S`	POSIX `[:…:]`
Program		Character Shorthands										Class Shorthands			
GNU *awk* Version 3.0.0	as `\y`	✓c	✓c	•	✓c	✓c	✓c	✓c	✓c	✓c▲	✓c▲	•	✓	•	✓c
GNU *sed* Version 2.05	•	•	•	•	•	✓	•	•	•	•	•	•	✓	•	✓c
Perl Version 5.003	✓	✓c	✓c	✓c	✓c	✓c	✓c	✓c	•	✓c	✓c	✓c	✓c	✓c	•
Tcl Version 7.5	•	√	√	•	√	√	√	√	√	√	√	•	•	•	•
GNU Emacs Version 19.33	✓	√	√	√	√	√	√	√	√	√	√	•	✓	✛	✛
Python Version 1.4β1	✓	√	√	•	√	√	√	√	√	√	√	•	✓	•	•
flex Version 2.5.1	•	✓c	✓c	•	✓c	✓c	✓c	✓c	✓c	✓c	✓c	•	•	•	✓c
GNU *egrep* Version 2.0	✓	•	•	•	•	•	•	•	•	•	•	•	✓	•	•

✓ supported √ pseudo support (☞75) ✓c also valid in a class (☞78)
✓▲ see text (☞75) ✛ not supported, but something similar is (☞78)

Generally, when you request one of these control characters in a regex or string, the byte you receive is the one provided by the C compiler for the same backslash escape. The C standards leave the selection of the actual values to the discretion of the compiler vendor.*

In practice, compilers for any particular platform are standardized on this point, so it's safe to view these as *operating-system dependent.* Also in practice, all but `\n` and `\r` are standardized across platforms—it's pretty safe to expect that `\t` gives you an ASCII tab pretty much everywhere you go where ASCII or a superset is used (which is pretty much everywhere I've ever been).

As the list on the previous page illustrates, though, `\n` and `\r` are unfortunately not quite standardized. For what it's worth, though, every platform in the standard GNU C distribution (which does not include MacOS) except the IBM 370 (which uses `EBCDIC`) maps `\n` and `\r` to ASCII linefeed and carriage return, respectively.

My advice is that when you want, for example, "a newline" for whatever system your script will happen to run on, use `\n`. When you need a specific byte value, such as when writing code for a defined protocol like HTTP,** use `\012` or whatever the standard calls for. (`\012` is an *octal escape*.)

---

* Many thanks to Hal Wine for enlightenment on these points.

** I've written a Perl command-line URL-fetcher, `webget` (available on my home page, see Appendix A). I got a lot of reports about it not working with certain hosts until I "dumbed it down" to not insist upon properly-formatted HTTP replies. It seems some web servers, in error, use `\n` rather than `\015\012` when generating their replies.

Octal escape \ *num*

Some implementations allow three-digit octal (base 8) escapes to be used to indicate a byte with a particular value. For example, ⌜`\015\012`⌟ matches an ASCII CR/LF sequence. Octal escapes can be convenient for inserting hard-to-type characters into an expression. In Perl, for instance, you can use ⌜`\e`⌟ for the ASCII escape character but you can't in *awk.* Since *awk* does support octal escapes, you can use the ASCII code for the escape character directly: ⌜`\033`⌟.

"Is 9 an octal digit?" and other oddities

Buggy implementations have provided ample opportunity for surprising inconsistencies. Except for very old versions of *lex,* there seems to be no trouble treating `\0079` properly. Octal escapes can be no longer than three digits, so `\0079` matches two characters: a byte with the octal value `7`, followed by a literal '9'. But what about `\079`? Most implementations realize that `9` is not an octal digit, so stop at `\07` for the octal escape, yielding the same results as `\0079` or `\79`. But *flex,* AT&T versions of *awk,* and Tcl, for example, consider `9` an octal digit (with the same value as `\11`)! Different still, GNU *awk* sometimes issues a fatal error.*

You might wonder what happens with out-of-range values like `\565` (8-bit octal values range from `\000` to `\377`). It seems that half the implementations leave it as a larger-than-byte value (which can never match), while the other half strip it to a byte (with this example, usually to `\165`, an ASCII `'u'`).

Hexadecimal escape \ *`x`num*

Similar to octal escapes, many utilities allow a hexadecimal (base 16) value to be entered using `\x`. ⌈`\x0D\x0A`⌋ matches the CR/LF sequence. The concerns noted in the previous paragraph are more complex with hexadecimal escapes. Some implementations allow only two-digit hexadecimal escapes, while some allow one-digit escapes as well. Still others allow any number of digits. This could be quite surprising if you expected that the hexadecimal escape in ⌈`ora`**`\x2E`**`com`⌋ to be `\x2E` rather than the `\x2ec` such implementations will understand.

You might think that once you learn the particulars of the utility, these kind of mistakes wouldn't happen. Unfortunately, when different implementations of the same tool act differently, the issues of portability and upgrading arise. For example, some versions of *awk* that I've seen, (GNU *awk* and MKS *awk*) read however many hexadecimal digits as are there, others (such as *mawk*) read only up to two. (AT&T *awk* reads up to two, too.)

---

* GNU *auwk* version 3.0.0, which was current at the time this book went to press, issues a fatal error. Like many bugs reported in this book, I have been told that it will be fixed in a future version.

Are these escapes literal?

If these escapes are supported, you might expect something like ⌈`[+\055*/]`⌋ to be a class to match a plus, minus (`055` is the ASCII code for `'-'`), star, or slash. With Perl, and most tools for that matter, it is just that—the thought being that if you go to the trouble to use an octal escape, you don't want its result to be interpreted as a metacharacter. Some implementations,[*] however, convert these escapes before the main regex engine examines the regex, so the engine sees the `'-'` that the escape was ostensibly supposed to hide. This results in `+-*` being treated as a range. This can be quite surprising, so I marked them with ▲: in Table 3-3.

Of all the regex engines I tested, only GNU and MKS *awk* actually do this kind of "dual-pass" processing. I say this even though I know that in the same situation, Tcl, Emacs, and Python will also consider the `\055` as a range metacharacter, but they don't get the ▲: mark. What's up?

Strings as Regular Expressions

Table 3-3 shows that regexes in Emacs, Tcl, and Python[**] support most of the character escapes listed. So why are they marked using ✓? Well, because in reality, the regex engines don't support those escapes at all, not even octal escapes. In these tools, regular-expression operands are usually provided as strings. This means that the language's normal string processing is done before the results are given to the regex engine to process as a regex. It is this string processing, not the regex engine, that supports the items marked with ✓. (We've already seen this with Emacs, on page 69.)

This means that you can use the escapes in regular expressions for most practical purposes, which is why I've listed them in Table 3-3. Remember, however, that the string processing that supports this happens only when the regular-expression operands are, indeed, strings. If they are provided via the command line, or perhaps read from a control file—provided any way other than as strings, the regex engine receives them "raw," and no such escapes are available. This is why, for example, they are not available in Emacs regular expressions entered directly by the user during an editing session.

What does all this mean to you? At a minimum, it means that these escapes can turn into metacharacters just like the ▲-marked items, and, more importantly, that if there is any conflict between string and regex metacharacters, the string

---

* For those wishing to be POSIX compliant, the standard is not easy to understand on this point. Paul Steinbach, at the time Product Manager at Mortice Kern Systems, was able to convince me that the latter interpretation, as screwy as it might seem, is the one mandated by POSIX.

** This entire section specifically does not apply to Perl. Perl's unique processing in this regard is discussed in detail in Chapter 7, starting on page 219.

metacharacters take precedence. To avoid misinterpretation, the conflicting regex metacharacters must be escaped.

⌈`\b`⌋ *as a backspace,* ⌈`\b`⌋ *as a word anchor*

You'll notice that Python and Emacs recognize `\b` in two ways. The ✓ version is for when `\b` is interpreted by the string processing, yielding a backspace. The ✔ version is for when `\b` is interpreted by the regex engine, yielding a word anchor.[*] To get the latter, you need `\\b` in the string—the string processing converts `\\` to a backslash, leaving `\b` for the regex engine. It's somewhat comical, but to create a regex subexpression to match a single literal backslash, you need *four* backslashes in your string. The regex needs to be ⌈`\\`⌋, and you need `\\` for each backslash you want to appear in the regex, so the grand total becomes `\\\\`.

Although Python and Emacs behave similarly for the cases mentioned so far, they differ in their treatment of backslash sequences that the string processing does not understand.

Emacs-style "escape is dropped" string processing

Like most regex packages and languages, Emacs strings drop unknown backslashes and include the subsequent character in the string. This means that you must escape every backslash that you want to find its way to the regex. We saw one example on page 68; here's another:

```
"\"[^\\\"]*\\(\\\\\\(.\\|\n\\)[^\\\"]*\\)*\""
```

Wow. Do you understand this at first glance? It's an example from real code,[**] but it's enough to make me dizzy. Since it is provided as a string, the regex engine won't actually see it until it goes through string processing; the regex engine will actually receive:

```
⌈"[^\"]*\(\\\(.\|␤\)[^\"]*\)*"⌋
```

That's a bit easier to understand, I think, but it might be useful to see it rewritten in a generic *egrep-style* flavor as a comparison:

```
⌈"[^\"]*(\\(.|NL)[^\"]*)*"⌋
```

It's a regex to match a doublequoted string. One note: a backslash is *not* a metacharacter inside an Emacs character class. This is similar to, say, *egrep,* but different from Perl, *lex,* and *awk.* In these latter tools, where a backslash can be special within a class, `[^\"]` must be written as `[^\\"]`. I'm getting ahead of myself, as we'll be seeing this regex in Chapters 4 and 5, but as a comparison, in Perl I would write this example as ⌈`(?s)"[^\\"]*(\\.[^\\"]*)*"`⌋.

---

* Perl, actually, supports both as well, but for completely different reasons. An anchor would make no sense within a character class, so `\b` is a shorthand for a backspace within a class.

** The entire standard Emacs *elisp* library is filled with regular expressions. This example comes from *hilit19.el.*

Python-style "escape is passed" string processing

Python strings take the opposite approach to unknown backslash escapes: they pass them through unmolested. Python *regexes* use ⌈`\1`⌋ through ⌈`\9`⌋ for the first nine backreferences, but ⌈`\v10`⌋ through ⌈`\v99`⌋ for the larger numbers. Python *strings* recognize `\v` as a vertical tab, so to actually refer to, say, parenthetical group 12, you need `\\v12` in your string. So far, this is just the same as with Emacs.

Python regexes recognize ⌈`\w`⌋ to match a word-constituent character (discussed momentarily in "Class Shorthands"). Because `\w` means nothing to Python strings, and Python strings pass through unrecognized escapes, you can safely use an unadorned `\w` in your string. This reduces the "backslashitis" that plagues Emacs, but it also limits the future growth potential of Python strings (which may well be considered a feature if you prize consistency above all else).

Class Shorthands, Dot, and Character Classes

Some tools provide a range of convenient shorthands for things you would otherwise write with a class:

`\d` **Digit** Generally the same as ⌈`[0-9]`⌋.

`\D` **Non-digit** Generally the same as ⌈`[^0-9]`⌋

`\w` **Part-of-word character** Often the same as ⌈`[a-zA-Z0-9]`⌋. Some (notably Perl, GNU *awk*, and GNU *sed*) include an underscore as well. GNU Emacs's `\w` can change meanings on the fly—see "syntax classes" below.

`\W`	**Non-word character** Generally the ⌜`[^…]`⌟ opposite of ⌜`\w`⌟.
`\s`	**Whitespace character** Often the same as ⌜`[•\f\n\r\t\v]`⌟.
`\S`	**Non-whitespace character** Generally the ⌜`[^…]`⌟ opposite of ⌜`\s`⌟.

These are also shown in Table 3-3. As described on page 65, a POSIX locale could influence the meaning of some of these shorthands. I know they can with Tcl, Emacs, and Perl—I'm sure there are others as well. It's best to check the documentation. (Even if you don't use locales, you might need to be aware of them for portability's sake.)

Emacs syntax classes

As an example of something quite different, GNU Emacs and relatives use ⌈`\s`⌋ to reference special "syntax classes". Here are two examples:

`\s`*char*	matches characters in the Emacs syntax class as described by *char*
`\S`*char*	matches characters not in the Emacs syntax class

⌈`\sw`⌋ matches a "word constituent" character (exactly the same as ⌈`\w`⌋), and ⌈`\s`⌋ matches a "whitespace character." Since this last one is so similar to Perl's ⌈`\s`⌋, I marked it with a ✚ in Table 3-3 to note its similar function.

These are special because exactly which characters fall into these classes can be modified on the fly, so, for example, the concept of which characters are word constituents can be changed depending upon what kind of program you're editing. (Details are in Chapter 6, starting on page 194.)

(Most) any character—dot

In some tools, dot is a shorthand for a character class that matches every possible character, while in others it is a shorthand to match every character *except a newline*. It's a subtle difference that is important when working with tools that allow target text to contain multiple logical lines (or to span logical lines, such as in a text editor).

The original Unix regex tools worked on a line-by-line basis, so the thought of matching a newline wasn't even an issue until the advent of *sed* and *lex*. By that time, ⌈`.*`⌋ had become a common idiom to match "the rest of the line," so disallowing it from crossing line boundaries kept it from becoming "too unwieldy."*

So, dot was made to match any character except a newline. Most modern tools allow multi-line target text, and seem split about 50/50 as which they choose to implement. (Also, see the related discussion of line versus string anchoring on page 81, and "Dot *vs.* a Negated Character Class" on the next page.) Somewhat less important for working with normal text, the POSIX standard dictates that dot not match a null (a byte with value `0`).

Character classes— `[…]` and `[^…]`

The basic concept of a character class has already been well covered, but let me emphasize again that the metacharacter rules change depending on whether you're in a character class or not. For example, in Table 3-3, only items marked with $\checkmark_{C}$ may be used in a class. (In effect, the ✓ items may be used as well, for the same reasons and with the same limitations as outlined on page 75.

---

* According to *ed's* author, Ken Thompson.

In many tools, the only metacharacters recognized within a character class are:

- a leading caret (to indicate a negated class)
- a closing bracket (to end the class)
- a dash as a range operator (such as to allow `0-9` as a convenient shorthand for `0123456789`)

In limited-metacharacter-class implementations, other metacharacters (including, in most tools, even backslashes) are not recognized. So, for example, you can't use `\-` or `\]` to insert a hyphen or a closing bracket into the class. (Conventions vary, but generally putting them first where they would otherwise make no sense as class metacharacters causes them to be interpreted as literals.)

In general, the order that characters are listed in a class makes no difference, and using ranges over a listing of characters is irrelevant to the execution speed (i.e., `[0-9]` is no different from `[9081726354]`).

A character class is always a *positive assertion.* In other words, it must always match a character to be successful. For a negated class, that character is one *not* listed. It might be convenient to consider a negated character class to be a "negated-list character class."

When using ranges, it is best to stay at or within ⌜`0-9`⌟, ⌜`a-z`⌟, or ⌜`A-Z`⌟, as these are "natural" ranges to work with. If you understand the underlying character encoding and decide that something like ⌜`.-m`⌟ suits your needs, it is still better to list the individual characters, as it is more easily understood. Of course, when dealing with binary data, ranges like ⌜`\x80-\xff`⌟ make perfect sense.

Dot vs. a negated character class

When working with tools that allow multi-line text to be searched, take care to note that dot usually does not match a newline, while a negated class like ⌜`[^"]`⌟ usually does. This could yield surprises when changing from something such as ⌜`".*"`⌟ to ⌜`"[^"]*`⌟. It's best to check on a tool-by-tool basis—the status of a few common utilities is shown in Table 3-4 on page 82.

POSIX bracket expressions

What we normally call a *character class*, the POSIX standard has decided to call a *bracket expression.* POSIX uses the term "character class" for a special feature used *within* a bracket expression.*

---

* In general, this book uses "character class" and "POSIX bracket expression" as synonyms to refer to the entire construct, while "POSIX character class" refers to the special range-like class feature described here.

POSIX bracket-expression "character class"

A POSIX character class is one of several special metasequences for use within a POSIX bracket expression. An example is `[:lower:]`, which represents any lowercase letter within the current locale (☞ 65). For normal English text, `[:lower:]` is comparable to `a-z`.

Since this entire sequence is valid only *within* a bracket expression, the full class comparable to ⌈`[a-z]`⌋ is ⌈`[[:lower:]]`⌋. Yes, it's that ugly. But it has the advantage of including other characters, such as ö, ñ, and the like (if the locale actually indicates that they are "lowercase letters").

The exact list of POSIX character classes is locale dependent, but the following, at least, are usually supported (and *must* be supported for full POSIX compliance):

`[:alnum:]`	alphabetic characters and numeric character
`[:alpha:]`	alphabetic characters
`[:blank:]`	space and tab
`[:cntrl:]`	control characters
`[:digit:]`	digits
`[:graph:]`	non-blank (not spaces, control characters, or the like)
`[:lower:]`	lowercase alphabetics
`[:print:]`	like `[:graph:]`, but includes the space character
`[:punct:]`	punctuation characters
`[:space:]`	all whitespace characters (`[:blank:]`, newline, carriage return, and the like)
`[:upper:]`	uppercase alphabetics
`[:xdigit:]`	digits allowed in a hexadecimal number (i.e., `0-9a-fA-F`).

It's not uncommon for a program that is not POSIX compliant to make some attempt to support these. I know that *flex* and GNU's *awk*, *grep*, and *sed* do (but, oddly, GNU *egrep* does not).

POSIX bracket-expression "character equivalents"

Some locales define *character equivalents* to indicate that certain characters should be considered identical for sorting and such. For example, a locale might define an equivalence class '`n`' as containing `n` and `ñ`, or perhaps one named '`a`' as containing `a`, g, and `á`. Using a notation similar to the `[:…:]` above, but with `'='` instead of a colon, you can reference these equivalence classes within a bracket expression: ⌜`[[=n=][=a=]]`⌟ matches any of the characters mentioned.

If a character equivalence with a single-letter name is used but not defined in the locale, it defaults to the collating sequence of the same name. Locales normally include all the normal characters as collating sequences —`[.a.]`, `[.b.]`, `[.c.]`, and so on—so in the absence of special equivalents, ⌜`[[=n=][=a=]]`⌟ defaults to ⌜`[na]`⌟.

POSIX bracket-expression "collating sequences"

As described on page 65, a locale can have *collating sequences* to describe how certain characters or sets of characters should be treated for sorting and such. A collating sequence that maps multiple physical characters to a single logical character, such as the `span-ll` example, is considered "one character" to a fully compliant POSIX regex engine. This means that something like ⌜`[^123]`⌟ match the `'ll'` sequence.

A collating sequence element can be included within a bracket expression using a `[.….]` notation: ⌜`torti[[.span-ll.]]a`⌟, matches `tortilla`. A collating sequence allows you to match against those characters that are made up of combinations of other characters. It also creates a situation where a bracket expression can match more than one physical character!

The other example, `eszet`, is merely a way to give an ordering to ⌟—it doesn't create a new logical character, so in a bracket-expression, `[.eszet.]` is just an odd way to write ⌟ (which is pretty odd in itself, unless you read German).

Having collating sequences also affects ranges. Since `span-ll` creates a logical character between `l` and `m`, the range `a-z` would include `'ll'`.

Anchoring

Anchoring metacharacters don't match actual text, but *positions* in the text. There are several common variations:

Start-of-line or start-of-string anchor—caret

Originally, caret meant "start of line," and *anchored* the expression to match only at the beginning of the line. For systems like *ed* and *grep,* where the text checked by a regex was always a whole line unit, there was no ambiguity between a logical line and simply "the text being checked." Other tools, however, often allow arbitrary text to be matched. If the text contains embedded newlines, you can think of the text as being composed of multiple logical lines. If so, should a caret match at the beginning of any logical line, or only at the start of the target text as a whole (what I've been calling the target *string*)?

Of course, the answer is "It depends." With a text editor, a "start of string" would be an effective "start of file," which would be pretty silly. On the other hand, *sed, awk,* and Tcl's carets match only at the beginning of the entire string. That string may be a single line, the whole file, or anything—how the data arrives to be checked is irrelevant to how it is checked. Perl can do both line- and string-oriented matches, but the default is that caret matches only at the beginning of the string. Table 3-4 on the next page summarizes caret and dollar for a few common utilities.

Table 3-4. String/Line Anchors, and Other Newline-Related Issues

Concern	***lex***	**Tcl**	***sed***	***awk***	**Perl**	**Python**	**Emacs**
^ matches at start of string as a whole	√	√	√	√	√	√	√
^ matches after any newline (i.e.. at the start of embedded logical lines)	√	•	•	•	•	√	√
$ matches at end of string as a whole	•	√	√	√	√	√	√
$ matches before string-ending newline	✓	•	•	•	√	✓	✓
$ matches before any newline (i.e., at end of embedded logical lines)	√	•	•	•	•	√	√
dot matches newline	•	√	√	√	•	•	•
negated class matches newline	√	√	√	√	√	√	√
notes: ✓ means "yes, but simply as an effect of matching before *any* newline." The meaning of Perl's dot and anchors can be changed. See "String Anchors" (☞ 232)							

Often, the whole distinction is irrelevant—almost all of these utilities normally access (and hence process) a file line by line, so the "line *vs.* text" distinction is usually a non-issue. However, once you start applying regexes to multi-line strings (however they might acquire their data), it matters.

Where in the regex is caret a metacharacter?

Within a character class there are separate rules completely, and most tools agree there, but outside a character class ^ is sometimes taken as an anchor metacharacter and sometimes as a literal caret. In most tools, it is a metacharacter when it "makes sense" (such as after ⌈(⌋ or ⌈\⌋), but with other tools, it is a metacharacter only when it appears at the start of the regex. In such a case, it is not what I would call a "first class" metacharacter.

End-of-line or end-of-string anchor   dollar

Dollar is the end-of-line/end-of-string counterpart to caret. It usually deals with the embedded newline/logical line problem the same way that caret does, but, as Table 3-4 illustrates, with a few additional twists. Some implementations match a dollar before any newline, while others do so only at the end of the string. Some match at the end of the string *or* before a string-ending newline. (Perl does this by default, but it can be changed to match before any newline.) Still others (notably *lex*) match only before a newline, but not in any other position (in particular, not at the end of the target text).

Word boundaries— `\<`... `\>` and `\b`, `\B`

Like caret and dollar, these match a location in the string. There are two distinct approaches. One provides ⌈`\<`⌋ and ⌈`\>`⌋ to match the beginning and ending location of a word. The other provides ⌈`\b`⌋ to match any word boundary (either a word beginning, or a word ending) and ⌈`\B`⌋ to match any position that's not a word boundary (which can be surprisingly useful when needed).

Each tool has its own idea of what constitutes a "word character," and those that support POSIX's locale (☞ 65) can vary with the locale. As presented on page 78, they can vary in Emacs as well, but for different reasons. In any case, word boundary tests are always a simple test of adjoining characters. No regex engine actually does linguistic analysis to decide about words: all consider "NE14AD8" to be a word, but not "M.I.T."

Grouping and Retrieving

`(`…`)` or `\(`…`\)`; `\1`, `\2`, `\3`, …

So far, I've given much more coverage to parentheses than to backreferences, a feature of a number of tools. If backreferences are supported, ⌜`\digit`⌟ refers to the actual text matched by the sub-expression within the digit*th* set of parentheses. (Count opening parentheses from the left to label sets.) Usually only those up to ⌜`\9`⌟ allowed, although some tools allow for any number.

As discussed earlier, some tools allow access to the text of ⌜`\1`⌟ , ⌜`F\2`⌟, and such outside the regex. Some allow access only within the replacement string of a substitution (usually via the same `\1`, `\2`, etc., but in that case they are replacement-string metacharacters, not regex metacharacters). Some tools allow access anywhere, such as with Perl's `$1` or Python's `MyRegex.group(l)`. Some tools allow access not only to the text of the matched subexpression, but also to the exact location of the match within the string. For advanced text processing needs, this is quite useful. GNU Emacs, Tcl, and Python are a few of the tools that offer this feature. (Notably absent from this list is Perl.)

Quantifiers

The quantifiers (star, plus, question mark, and intervals—metacharacters that affect the *quantity* of what they govern) have been well discussed already. However, note that in some tools, ⌜`\+`⌟ and ⌜`\?`⌟ are used instead of ⌜`+`⌟ and ⌜`?`⌟. Also, with some tools, quantifiers can't be applied to a backreference, or to a set of parentheses.

As an example of something very different, Perl offers the ungainly looking `*?`, `+?`, `??`, and `{min,max}?` (constructs that are generally illegal in other flavors). They are the *non-greedy*[*] versions of the quantifiers. Quantifiers are normally "greedy," and try to match as much as possible. Perl provides these greedy quantifiers, but it also offers the non-greedy versions which, conversely, match as little as they can. The next chapter explains all the details.

---

[*] Also called *minimal matching* and *lazy,* among others (☞ 225).

Intervals `{min,max}` or `\{min,max\}`

Intervals can be considered a "counting quantifier" because you specify exactly the minimum number of matches you wish to require and the maximum number of matches to allow. If only a single number is given (such as in ⌜`[a-z]{3}`⌟ or ⌜`[a-z]\{3\}`⌟, depending upon the flavor), and if supported by the tool, it matches exactly that many of the item. This example is the same as ⌜`[a-z][a-z][a-z]`⌟, although the latter can be more efficient with some types of engines (☞ 156).

One caution: don't think you can use something like ⌜`x{0,0}`⌟ to mean "there must not be an `x` here." ⌜`x{0,0}`⌟, is a meaningless expression because it means "no *requirement* to match ⌜`x`⌟, and, in fact, don't even bother trying to match any. Period. " It's the same as if the whole ⌜`X{0,0)`⌟ wasn't there at all —if there is an `x` present, it could still be matched by something later in the expression, so your intended purpose is defeated.*

Since the undesired item in this example is a single character, you use ⌜`[^x]`⌟ to indicate that a character must be there, but that it must be anything except `x`. This might work for some needs, but it is still quite different from "ensure no `x` at this point" since ⌜`[^x]`⌟ actually requires a character in order to match. The concept of `x` not existing doesn't require any character. The only popular regex flavor that provides this functionality is Perl (☞ 228).

Alternation

Alternation allows any one of several subexpressions to match at a given point. Each subexpression is called an *alternative.* The ⌜`|`⌟ symbol is called various things, but *or* and *bar* seem popular (some flavors use ⌜`\|`⌟ instead).

Alternation is always a high-level construct (one that has very low precedence). This means that ⌜`this and|or that`⌟ is the same as ⌜`(this and)|(or that)`⌟, not the ⌜`this(and|or)that`⌟ that is probably much more useful. One exception is that the line anchors in *lex* are not first-class —they are valid only at the edges of the regex, and have even less precedence than alternation. This means that in *lex*, ⌜`^this|that$`⌟ is the same as ⌜`^(this|that)$`⌟, not the ⌜`(^this)|(that$)`⌟ that it means pretty much everywhere else.

Although the POSIX standard, *lex*, and most versions of *awk* disallow something like ⌜`(this|that|)`⌟ with an empty alternative, I think it's certainly reasonable to want to use it. The empty subexpression means to always match, so, this example is logically comparable to ⌜`(this|that)?`⌟. That's in theory; in practice, what happens is different for most tools. *awk, lex,* and *egrep* are among the few tools for which it would actually be identical   this is the topic of Chapter 4. Even if it were exactly the same, it could be useful for its notational convenience or clarity. As Larry Wall explained to me once, "It's like having a zero in your numbering system."

---

* In theory, what I say about `{0,0}` is correct. In practice, what actually happens is even worse—it's almost random! In many programs (including GNU *awk*, GNU *grep*, and older versions of Perl) it seems that `{0,0}` means the same as `*`, while in many others (including most versions of *sed* that I've seen, and some versions of *grep)* it means the same as `?`. Crazy!

Guide to the Advanced Chapters

Now that we're familiar with metacharacters, flavors, syntactic packaging, and the like, it's time to start getting into the nitty-gritty details, the meat of the subject, the "hard core," if you will. This begins in Chapter 4, *The Mechanics of Expression Processing.* There are different ways a tool's match engine might be implemented, and the selection influences *if* a match is achieved or not, *which* text in the string gets matched, and *how much time* the whole thing takes. We'll look at all the details. As a byproduct of this knowledge, it will become much easier for you to craft complex expressions with confidence.

This brings us to Chapter 5, *Crafting a Regular Expression.* Once you know the basics about how an engine works, you can consider techniques to take full advantage of that knowledge   and of the engine. Chapter 5 looks at some of the pitfalls of certain popular regex flavors   pitfalls that often lead to unwelcome surprises   and turns the tables to put them to use for us.

Chapters 4 and 5 are the central core of this book, its essence. These first three chapters merely lead up to them, and the discussions in the tool-specific chapters that follow rely on them. It's not necessarily what you would call light reading, but I've taken great care to stay away from math and algebra and all that stuff that's just mumbo-jumbo to most of us. As with any large amount of new information, it will likely take time to sink in and internalize.

Tool-Specific Information

The lessons and techniques discussed in Chapters 4 and 5 transcend any particular tool. You might need to cosmetically adjust the examples to fit a certain flavor, but it's the lessons that are important, not the examples.

Languages such as *awk*, Tcl, Python, *sed*, and Emacs have consistent interfaces to their regular expressions. Once you learn how their engine works (Chapters 4 and 5), there's not much tool-specific information to discuss except a few comments about its particular Shine and Finish. We look at this in Chapter 6.

On the other hand, Perl and *The Perl Way* are linked to regular expressions on many levels. Perl's rich and expressive interface to regular expressions has many nooks and crannies to explore. Some consider Perl to be the Muscle Car of scripting languages, and some consider it the Funny Car. The same features that allow an expert to solve Fermat's last theorem* with a one-liner are the same features that can create a minefield for the unaware. For these reasons, Chapter 7 puts Perl regular expressions and operators under the microscope. General lessons on Perl programming are sprinkled throughout, but the prime focus is on understanding and exploiting Perl's many regex-related features.

---

* I'm exaggerating, of course, but if you'd like to try it, see: `http://www.yahoo.com/Science/Mathematics/Problems/Fermat_s_Last_Theorem/`

4

The Mechanics of Expression Processing

In this chapter:

- ***Start Your Engines!***
- ***Match Basics***
- ***Regex-Directed vs. Text***
- ***Backtracking***
- ***More About Greediness***
- ***NFA, DFA, and POSIX***
- ***Practical Regex Techniques***
- ***Summary***

Now that we have some background under our belt, let's delve into the mechanics of how a regex engine really goes about its work. Here we don't care much about the Shine and Finish of the previous chapter; this chapter is all about the engine and the drive train, the stuff that grease monkeys talk about in bars. We'll spend a fair amount of time under the hood, so expect to get a bit dirty with some practical hands-on experience.

Start Your Engines!

Let's see how much I can milk this engine analogy for. The whole point of having an engine is so that you can get from Point A to Point B without doing much work. The engine does the work for you so you can relax and enjoy the Rich Corinthian Leather. The engine's primary task is to turn the wheels, and how it does that isn't really a concern of yours. Or is it?

Two Kinds of Engines

Well, what if you had an electric car? They've been around for a long time, but they aren't as common as gas cars because they're hard to design well. If you had one, though, you would have to remember not to put gas in it. If you had a gasoline engine, well, watch out for sparks! An electric engine more or less just runs, but a gas engine might need some babysitting. You can get much better performance just by changing little things like your spark plug gaps, air filter, or brand of gas. Do it wrong and the engine's performance deteriorates, or, worse yet, it stalls.

Each engine might do its work differently, but the end result is that the wheels turn. You still have to steer properly if you want to get anywhere, but that's an entirely different issue.

New Standards

Let's stoke the fire by adding another variable: the California Emissions Standards.* Some engines adhere to California's strict pollution standards, and some engines don't. These aren't really different kinds of engines, just new variations on what's already around. The standard regulates a result of the engine's work, the emissions, but doesn't say one thing or the other about how the engine should go about achieving those cleaner results. So, our two classes of engine are divided into four types: electric (adhering and non-adhering) and gasoline (adhering and non-adhering).

Come to think of it, I bet that an electric engine can qualify for the standard without much change, so it's not really impacted very much —t he standard just "blesses" the clean results that are already par for the course. The gas engine, on the other hand, needs some major tweaking and a bit of re-tooling before it can qualify. Owners of this kind of engine need to pay *particular* care to what they feed it use the wrong kind of gas and you're in big trouble in more ways than one.

The impact of standards

Better pollution standards are a good thing, but they require that the driver exercise more thought and foresight (well, at least for gas engines, as I noted in the previous paragraph). Frankly, however, the standard doesn't impact most people since all the other states still do their own thing and don't follow California's standard . . . yet. It's probably just a matter of time.

Okay, so you realize that these four types of engines can be classified into three groups (the two kinds for gas, and electric in general). You know about the differences, and that in the end they all still turn the wheels. What you don't know is what the heck this has to do with regular expressions! More than you might imagine.

Regex Engine Types

There are two fundamentally different types of regex engines: one called "DFA" (the electric engine of our story) and one called "NFA" (the gas engine). The details follow shortly (☞ 101), but for now just consider them names, like Bill and Ted. Or electric and gas.

Both engine types have been around for a long time, but like its gasoline counterpart, the NFA type seems to be used more often. Tools that use an NFA engine

---

* California has rather strict standards regulating the amount of pollution a car can produce. Because of this, many cars sold in America come in "for California" and "non-California" models.

include Tcl, Perl, Python, GNU Emacs, *ed, sed, vi,* most versions of *grep,* and even a few versions of *egrep* and *awk.* On the other hand, a DFA engine is found in almost all versions of *egrep* and *awk,* as well as *lex* and *flex.* Table 4-1 on the next page lists a few common programs available for a wide variety of platforms and the regex engine that most versions use. A *generic* version means that it's an old tool with many clones—I have listed notably different clones that I'm aware of.*

As Chapter 3 illustrated, 20 years of development with both DFAs and NFAs resulted in a lot of needless variety. Things were dirty. The POSIX standard came in to clean things up by specifying clearly which metacharacters an engine should support, and exactly the results you could expect from them. Superficial details aside, the DFAs (our electric engines) were already well suited to adhere to the standard, but the kind of results an NFA traditionally provided were quite different from the new standard, so changes were needed. As a result, broadly speaking, we have three types of regex engines:

- DFA (POSIX or not—similar either way)
- Traditional NFA
- POSIX NFA

POSIX standardized the workings of over 70 programs, including traditional regex-wielding tools such as *awk, ed, egrep, expr, grep, lex,* and *sed.* Most of these tools' regex flavor had (and still have) the weak powers equivalent to a moped. So weak, in fact, that I don't find them interesting for discussing regular expressions. Although they can certainly be extremely useful tools, you won't find much mention of *expr, ed,* and *sed* in this book. Well, to be fair, some modern versions of these tools have been retrofitted with a more-powerful flavor. This is commonly done to *grep,* a direct regex sibling of *sed, ed,* and *expr.*

On the other hand, *egrep, awk,* and *lex* were normally implemented with the electric DFA engine, so the new standard primarily just confirmed the status quo—no big changes. However, there *were* some gas-powered versions of these programs which had to be changed if they wanted to be POSIX-compliant. The gas engines that passed the California Emission Standards tests (POSIX NFA) were fine in that they produced results according to the standard, but the necessary changes only enhanced their fickleness to proper tuning. Where before you might get by with slightly misaligned spark plugs, you now have absolutely no tolerance. Gasoline that used to be "good enough" now causes knocks and pings. But so long as you know how to maintain your baby, the engine runs smoothly. And cleanly.

---

* Where I could find them, I used comments in the source code to identify the author (or, for the *generic* tools, the original author). I relied heavily on Libes and Ressler's *Life With Unix* (Prentice Hall, 1989) to fill in the gaps.

Table 4-1: Some Tools and Their Regex Engines

Program	(Original) Author	Version	Regex Engine
awk	Aho, Weinberger, Kernighan	*generic*	DFA
new *awk*	Brian Kernighan	*generic*	DFA
GNU *awk*	Arnold Robbins	*recent*	Mostly DFA, some NFA
MKS *awk*	Mortice Kern Systems		POSIX NFA
mawk	Mike Brennan	*all*	POSIX NFA
egrep	Alfred Aho	*generic*	DFA
MKS *egrep*	Mortice Kern Systems		POSIX NFA
GNU Emacs	Richard Stallman	*all*	Trad. NFA (POSIX NFA available)
Expect	Don Libes	*all*	Traditional NFA
expr	Dick Haight	*generic*	Traditional NFA
grep	Ken Thompson	*generic*	Traditional NFA
GNU *grep*	Mike Haertel	Version 2.0	Mostly DFA, but some NFA
GNU *find*	GNU		Traditional NFA
lex	Mike Lesk	*generic*	DFA
flex	Vern Paxson	*all*	DFA
lex	Mortice Kern Systems		POSIX NFA
more	Eric Schienbrood	*generic*	Traditional NFA
less	Mark Nudelman		Variable (usually Trad. NFA)
Perl	Larry Wall	*all*	Traditional NFA
Python	Guido van Rossum	*all*	Traditional NFA
sed	Lee McMahon	*generic*	Traditional NFA
Tcl	John Ousterhout	*all*	Traditional NFA
vi	Bill Joy	*generic*	Traditional NFA

From the Department of Redundancy Department

At this point I'll ask you to go back and review the story about engines. Every sentence there rings with some truth about regular expressions. A second reading should raise some questions. Particularly, what does it mean that an electric DFA more or less "just runs?" What kind of things affect a gas-powered NFA? How can I tune my NFA? What special concerns does an emissions-controlled **POSIX** NFA have? What's a "stalled engine" in the regex world? Last, and certainly least, just what is the regex counterpart to Rich Corinthian Leather?

Match Basics

Before looking at the differences among these engine types, let's first look at their similarities. Certain aspects of the drive train are the same (or for all practical purposes appear to be the same), so these examples can cover all engine types.

About the Examples

This chapter is primarily concerned with a generic, full-function regex engine, so some tools won't support exactly everything presented. In my examples, the dip stick might be to the left of the oil filter, while under your hood it might be behind the distributor cap. Your goal is to understand the concepts so that you can drive and maintain your favorite regex package (and ones you find interest in later).

I'll continue to use Perl's notation for most of the examples, although I'll occasionally show others to remind you that the *notation* is superficial and that the issues under discussion transcend any one tool or flavor. To cut down on wordiness here, I'll rely on you to check Chapter 3 if I use an unfamiliar construct.

This chapter details the practical effects of how a match is carried out. It would be nice if everything could be distilled down to a few simple rules that could be memorized without needing to understand what is going on. Unfortunately, that's not the case. In fact, with all this chapter offers, I identify only two all-encompassing rules I can list for you:

1. the earliest match wins

2. the quantifiers are greedy

We'll look at these rules, their effects, and much more throughout this chapter. Let's start by diving into the details of the first rule.

Rule 1: The Earliest Match Wins

Let's get down to business with *The First Rule of Regular Expressions:*

> *The match that begins earliest wins*

This rule says that any match that begins earlier in the string is always preferred over any plausible match that begins later. This rule doesn't say anything about how long the winning match might be (we'll get into that shortly), merely that among all the matches possible anywhere in the string, the one that begins the leftmost in the string is chosen. Actually, since more than one plausible match can start at the same earliest point, perhaps the rule should read "*a* match. . . " instead of "*the* match. . . ," but that sounds odd.

Here's how the rule comes about: the match is first attempted at the very beginning (just before the first character) of the string to be searched. "Attempted" means that every permutation of the entire (perhaps complex) regex is tested starting right at that spot. If all possibilities are exhausted and a match is not found, the complete expression is re-tried starting from just before the second character. This full retry occurs at each position in the string until a match is found. A "no match" result is reported only if no match is found after the full retry has been attempted at each position all the way to the end of the string (just after the last character).

Thus, when trying to match ⌜`ORA`⌟ against `FLORAL`, the first attempt at the start of the string fails (since ⌜`ORA`⌟ can't match `FLO`). The attempt starting at the second character also fails (it doesn't match `LOR` either). The attempt starting at the third position, however, does match, so the engine stops and reports the match: `FLORAL`.

If you didn't know this rule, results might sometimes surprise you. For example, when matching ⌜`cat`⌟ against

```
    The dragging belly indicates your cat is too fat
```

the match is in `indicates`, not at the word cat that appears later in the line. This word cat *could* match, but the `cat` in `indicate` appears earlier in the string, so it is the one that is matched. For an application like *egrep* which cares only *whether* there is a match, not *where* the match might be, the distinction is irrelevant. For other uses, such as with a search-and-replace, it becomes paramount.

Remember, the regex is tried *completely* each time, so something such as ⌜`fat|cat|belly|your`⌟ matches `'belly'`

```
    The dragging belly indicates your cat is too fat
```

rather than fat, even though ⌜`fat`⌟ is listed first among the alternatives. Sure, the regex could conceivably match fat and the others, but since they are not the *earliest* (starting furthest to the left) possible match, they are not the one chosen. As I said, the entire regex is attempted completely from one spot before moving along the string to try again from the next spot, and in this case that means trying each alternative ⌜`fat`⌟, ⌜`cat`⌟, ⌜`belly`⌟, and ⌜`your`⌟ at each position before moving on.

The "Transmission" and the Bump-Along

It might help to think of this rule as the car's transmission, connecting the engine to the drive train while adjusting for the gear you're in (or changing gears for you if it's an automatic—perhaps the automotive equivalent of some internal optimizations we'll be talking about in the next chapter). The engine itself does the real work (turning the crank); the transmission transfers this work to the wheels.

The transmission's main work: the bump-along

If the engine can't find a match starting at the beginning of the string, it's the transmission that bumps the regex engine along to attempt a match at the next position in the string, and the next, and the next, and so on. Usually. If, for instance, a regex begins with a start-of-string anchor, the transmission can realize that any bump-along would be futile, for only the attempt at the start of the string could possibly be successful. This is an example of the "String/Line Anchors" optimization discussed in the next chapter (☞ 158).

Engine Pieces and Parts

An engine is made up of parts of various types and sizes. You can't possibly hope to understand how the whole thing works if you don't know much about the individual parts. In a regex, these parts are the individual units—literal characters, quantifiers (star and friends), character classes, parentheses, and so on. The combination of these parts (and the engine's treatment of them) makes a regex what it is, so looking at ways they can be combined and how they interact is our primary interest. First, let's take a look at some of the individual parts:

Literal text

With a literal, non-metacharacter like ⌈`z`⌋ or ⌈`!`⌋, the match attempt is simply "Does this literal character match the current text character?" If your regex is only literal text, such as ⌈`usa`⌋, it is treated as " ⌈`u`⌋ and then ⌈`s`⌋ and then ⌈`a`⌋. " It's a bit more complicated if you have the engine to do a case-insensitive match, where ⌈`b`⌋ matches `B` and vice-versa, but it's still pretty straightforward.

Character classes, dot, and the like

Matching a character class is not difficult either. Regardless of the length of the character class, it still matches just one character.* A character class represents a set of characters that can match. Characters are included explicitly, or in a negated class excluded explicitly. Dot is just a shorthand for a large character class that matches any character (any character except newline and/or null in some flavors), so it's not a problem either. The same applies to other shorthand conveniences such as ⌈`\w`⌋, ⌈`\W`⌋, ⌈`\d`⌋, ⌈`\D`⌋, ⌈`\s`⌋, ⌈`\S`⌋, and the like.

Anchors

A few other metacharacters are almost as simple, but they don't actually match characters in the target string, rather, they match a position in the target string. This includes string/line boundaries (caret and dollar), as well as word boundaries ⌈`\<`⌋, ⌈`\b`⌋, and such. The tests are simple because, for the most part, they simply compare two adjacent characters in the target string.

Simple parentheses

Certain parentheses used only for capturing text, as opposed to those used merely for grouping, have some performance impact (discussed in Chapter 5), but for the most part, they don't change how the match is carried out.

No electric parentheses or backreferences

I'd like to first concentrate on the similarities among the engines, but as foreshadowing, I'll show an interesting difference. Capturing parentheses (and the associated backreferences) are like a gas additive—they affect a gasoline engine, but they are irrelevant to an electric engine because it doesn't use gas in the first place. The way a DFA engine works completely precludes the concept of backreferences and capturing parentheses. It just can't happen.* This explains why tools developed with DFAs don't provide these features. You'll notice that *awk, lex,* and *egrep* don't have backreferences or any `$1` type functionality.

---

* Actually, as we saw in the previous chapter (☞81), a POSIX collating sequence *can* match multiple characters, but this is not common.

You might, however, notice that GNU's version of *egrep* **does** support backreferences. It does so by having two complete engines under the hood! It first uses a DFA engine to see whether a match is likely, and then uses an NFA engine (which supports the full flavor, including backreferences) to confirm the match. Later in this chapter, we'll see why a DFA engine can't deal with backreferences or capturing, and why anyone ever bothers with such an engine at all. (It has some major advantages, such as being able to match very quickly.)

Rule 2: Some Metacharacters Are Greedy

So far, we've seen matching that is quite straightforward. It is also rather uninteresting—you can't *do* much without involving more-powerful metacharacters such as star, plus, alternation, and so on. Their added power requires more information to understand them fully.

First, you need to know that the quantifiers (`?`, `*`, `+`, and `{`*min, max*`}`) are *greedy.* When one of these governs a subexpression, such as the `a` in ⌈`a?`⌋, the ⌈`(expr)`⌋ in ⌈`(expr)*`⌋, or the ⌈`[0-9]`⌋ in ⌈`[0-9]+`⌋, there is a minimum number of matches that are required before it can be considered successful, and a maximum number that it will ever attempt to match. This has been mentioned in earlier chapters—what's new here concerns *The Second Rule of Regular Expressions:*

> Items that are allowed to match a variable number of times always attempt to match as much as possible. *They are greedy.*

In other words, the quantifiers settle for the minimum number of required matches *if they have to,* but they always attempt to match as many times as they can, up to their maximum allowed.

The only time they settle for anything less than their maximum allowed is when matching too much ends up causing some later part of the regex to fail. A simple example is using ⌜`\<\w+s\>`⌟ to match words ending with an '`s`', such as `regexes`. The ⌜`\w+`⌟ alone is happy to match the entire word, but if it does, the ⌜`s`⌟ can not match. To achieve the overall match, the ⌜`\w+`⌟ settles for matching `regexes`, thereby allowing ⌜`s\>`⌟, and thus the full regex, to match.

---

* This does not, of course, mean that there can't be some mixing of technologies to try to get the best of both worlds. This is discussed in a sidebar on page 121.

If it turns out that the only way the rest of the regex can succeed is when the greedy construct in question matches nothing (and if zero matches are allowed, as with star, question, and `{0,`*max*`}` intervals), well, that's perfectly fine too. However, it turns out this way only if the requirements of some later subexpression forces the issue. Because the quantifiers always (or, at least, try to) take more than they minimally need, they are called greedy.

This rule has many useful (but sometimes troublesome) implications. It explains, for example, why ⌈`[0-9]+`⌋ matches the full number in `March•1998`. Once the `1` has been matched, the plus has fulfilled its minimum requirement, but because it tries for its maximum number of matches, it continues and matches the `998` before being forced to stop by the end of the string. (Since ⌈`[0-9]`⌋ can't match the nothingness at the end of the string, the plus finally stops.)

A subjective example

Of course, this method of grabbing things is useful for more than just numbers. Let's say you have a line from an email header and want to check whether it is the subject line. As we saw in earlier chapters, you simply use ⌈`^Subject:•`⌋. However, if you use ⌈`^Subject:•(.*)`⌋, you can later access the text of the subject itself via the tool's after-the-fact parenthesis memory (for example, `$1` in Perl).*

Before looking at why ⌈`.*`⌋ matches the entire subject, be sure to understand that once the ⌈`^Subject:•`⌋ part matches, you're guaranteed that the entire regular expression will eventually match. You know this because there's nothing after ⌈`^Subject:•`⌋ that could cause the expression to fail ⌈`.*`⌋ can *never* fail since the worst case of "no matches" is still considered successful for star.

Why do we even bother adding ⌈`.*`⌋? Well, we know that because star is greedy, it attempts to match dot as many times as possible, so we use it to "fill" `$1`. In fact, the parentheses add nothing to the logic of what the regular expression matches in this case we use them simply to capture the text matched by ⌈`.*`⌋.

Once ⌜`.*`⌟ hits the end of the string, the dot isn't able to match, so the star finally stops and lets the next item in the regular expression attempt to match (for even though the starred dot could match no further, perhaps a subexpression later in the regex could). Ah, but since it turns out that there is no next item, we reach the end of the regex and we know that we have a successful match.

---

* This example uses capturing as a forum for presenting greediness, so the example itself is appropriate only for NFAs (because only NFAs support capturing). The lessons on greediness, however, apply to all engines, including the non-capturing DFA.

So, with a variable `$line` holding a string such as

```
Subject: Re: happy birthday
```

the Perl code

```
if ( $line =~ m/^Subject: (.*)/) {
   print "The subject is: $1\n";
}
```

produces `'The subject is: Re: happy birthday'`.

To make the example more concrete, here's the snippet in Tcl

```
if [regexp "^Subject: (.*)" $line all exp1] {
   puts "The subject is: $exp1"
}
```

and in Python:

```
reg = regex.compile("Subject: \(.*\)")
if reg.match(line) > 0:
    print "The subject is:", reg.group(l)
```

As you can see, each language handles regular expressions in its own way, but the concept (and result) of greediness stays the same.

Regarding replies

To extend this example, let's consider bypassing that pesky `'Re:•'` that most mail systems prepend to a subject to indicate the message is a reply. Our goal is to ignore any `'Re:•'` that might begin the subject, printing only the "real" subject part.

We can use greediness to our advantage and take care of this right in the regex. Consider ⌈`^Subject:•(Re:•)?(.*)`⌋, with ⌈`(Re:)?`⌋ added before ⌈`(.*)`⌋. Both subexpressions are greedy, but ⌈`(Re:•)?`⌋ gets to be greedy *first*, nabbing `'Re:•'` before ⌈`.*`⌋ takes what's left. In fact, we could use ⌈`(Re:•)*`⌈ just as well, which scoops up all the `Re:` that might have been stacked up over the course of back and forth replies.

The parentheses in ⌈`(Re:•)?`⌋ are intended only for grouping, but they still count as a set of parentheses. This means that our original parentheses, which grab what ⌈`.*`⌋ matches, become the second set. This, in turn, means that the subject of our interest becomes `$2`, not `$1`. So, if our code is

```
    if ( $line =~ m/^Subject: (Re: )?(.*)/ ) {
        print "The subject is: $2\n";
    }
```

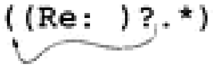
we get `'The subject is: happy birthday'`.

You might even imagine something like:

```
if ( $line =~ m/^Subject: (Re: )?(.*)/ ) {
   #  we have a match -- alter our report depending upon $1
   if ($1 eq "Re: ") {
      print "The reply is: $2\n";
   } else {
      print "The subject is: $2\n";
   }
}
```

Even if the set of parentheses that fills `$1` is not able to match, the second set still stuffs its matched text into `$2`.

As a final comparison, let's look at the same expression with one of the parentheses moved:

`^Subject:` 

Since sets of parentheses are always labeled by counting open parentheses from the left, the ⌈`'Re:•`⌋ parentheses become `$2`, and the whole subject becomes `$1`. However, this time the `'Re:•'` that might be matched into `$2` is also within the first set of parentheses, so `$1` will have that same `'Re:•'` (as well as the rest of the subject). Although this isn't useful with our example so far, it would be if you wanted to access the subject with any `'Re:•'` intact, but also want a simple way to tell whether it is a reply.

Being too greedy

Let's get back to the concept of a greedy quantifier being as greedy as it can be. Consider how the matching and results would change if we add another ⌜`.*`⌟: ⌜`^Subject:•(.*).*`⌟. The answer is: nothing would change. The initial ⌜`.*`⌟ (inside the parentheses) is so greedy that it matches all the subject text, never leaving anything for the second ⌜`.*`⌟ to match. Again, failing to match is not a problem, since the star does not require a match to be successful. Were the second ⌜`.*`⌟ in parentheses as well, the resulting `$2` would always be empty.

Does this mean that after ⌜`.*`⌟, a regular expression can never have anything that is expected to actually match? No, of course not. It is possible for something later in the regex to *force* something previously greedy to give something back (that is, relinquish or "unmatch") if that's what is necessary to achieve an overall match.

Let's consider the possibly useful ⌜`^.*([0-9][0-9])`⌟, which matches the *last* two digits in the string, wherever they might be, and saves them to `$1`. Here's how it works: At first, ⌜`.*`⌟ matches the entire string. Because the following ⌜`([0-9][0-9])`⌟ *is required*, its failure to match, in effect, tells ⌜`.*`⌟ "Hey, you took too much! Give me back something so that I can have a chance to match." Greedy components first try to take as much as they can, but they *always* defer to the greater need to achieve an overall match. They're just stubborn about it, and only do so when forced. Of course, they'll never give up something that hadn't been optional in the first place, such as a plus quantifier's first match.

With this in mind, let's apply ⌈`^.*([0-9][0-9])`⌋ to `'about•24•characters•long'`. Once ⌈`.*`⌋ matches the whole string, the requirement for the first ⌈`[0-9]`⌋ to match forces ⌈`.*`⌋ to give up '`g`' (the last thing it had matched). That doesn't, however, allow ⌈`[0-9]`⌋ to match, so ⌈`.*`⌋ is again forced to relinquish something, this time the `n` in long. This cycle continues 15 more times until ⌈`.*`⌋ finally gets around to giving up `4`.

Unfortunately, even though the first ⌈`[0-9]`⌋ can then match, the second still cannot. So, ⌈`.*`⌋ is forced to relinquish more in search of the overall match. This time ⌈`.*`⌋ gives up the `2`, which the first ⌈`[0-9]`⌋ can match. Now, the `4` is free for the second ⌈`[0-9]`⌋ to match, and the entire expression matches '`about•24•char`…', with `$1` getting '`24`'.

First come, first served

Consider changing that regex to ⌈`^.*([0-9]+)`⌋, ostensibly to match not just the last two digits, but the last whole number however long it might be. It won't work. As before, ⌈`.*`⌋ is forced to relinquish some of what it had matched because the subsequent ⌈`[0-9]+`⌋ requires a match to be successful. With the '`about•24•char`…' example, that means unmatching until the `4`. Like before, ⌈`[0-9]`⌋ can then match. *Unlike* before, there's then nothing further that *must* match, so ⌈`.*`⌋ is not forced to give up the `2` or any other digits it might have matched. Were ⌈`.*`⌋ to do so, the ⌈`[0-9]+`⌋ would certainly be a grateful recipient, but nope, first come first served. Greedy constructs are very possessive—once something is in their clutches, they'll give it up if forced, but never just to be nice.

If this feels counter-intuitive, realize that ⌈`[0-9]+`⌋ is just one match away from ⌈`[0-9]*`⌋, which is in the same league as ⌈`.*`⌋. Substituting into ⌈`^.*([0-9]+)`⌋ we get ⌈`^.*(.*)`⌋ as our regex, which looks suspiciously like the ⌈`^Subject:•(.*).*`⌋ from a page or so ago.

Getting down to the details

I should clear up a few things here. Phrases like "the ⌈`.*`⌋ *gives up* . . ." and "the ⌈`[0-9]`⌋ *forces* . . ." are slightly misleading. I used these terms because they're easy to grasp, and the end result appears to be the same as reality. However, what really happens behind the scenes depends on the basic engine type, DFA or NFA. So it's time to see what these really are.

Regex-Directed vs. Text-Directed

The two basic engine types reflect a fundamental difference in how one might apply a regular expression in checking a string. I call the gasoline-driven NFA engine "`regex-directed,`" and the electric-driven DFA "`text-directed.`"

NFA Engine: Regex-Directed

Let's consider one way an engine might match ⌈`to(nite|knight|night)`⌋, against the text '…`tonight`…' Starting with the ⌈`t`⌋, the regular expression is examined one component at a time, and the "current text" is checked to see whether it matches. If it does, the next component is checked, and so on, until all components have matched, indicating that an overall match has been achieved.

With the ⌈`to(nite|knight|night)`⌋ example, the first component is ⌈`t`⌋, which repeatedly fails until a `t` is reached. Once that happens, the ⌈`o`⌋ is checked against the next character, and if it matches, control moves to the next component. In this case, the "`next component`" is '⌈`(nite|knight|night)`⌋ which really means "⌈`nite`⌋ or ⌈`knight`⌋ or ⌈`night`⌋." Faced with three possibilities, the engine just tries each in turn. We (humans with advanced neural nets between our ears) can see that if we're matching tonight, the third alternative is the one that leads to a match. Despite their brainy origins (☞ 60), a regex-directed engine can't come to that conclusion until actually going through the motions to check.

Attempting the first alternative, ⌈`nite`⌋, involves the same component-at-a-time treatment as before: "Try to match ⌈`n`⌋, then ⌈`i`⌋, then ⌈`t`⌋, and finally ⌈`e`⌋." If this fails, as it eventually does, the engine tries another alternative, and so on until it achieves a match or must report failure. Control moves within the regex from component to component, so I call it "`regex-directed.`"

The control benefits of an NFA engine

In essence, each subexpression of a regex in a regex-directed match is checked independently of the others —their only interrelation is that of proximity to one another by virtue of being subexpressions thrown together to make a single regex. The layout of the components is what controls an engine's overall movement through a match.

Since the regex directs the NFA engine, the driver (the writer of the regular expression) has considerable opportunity to craft just what he or she wants to happen. (Chapter 5 is devoted to this very subject.) What this really means may seem vague now, but it will all be spelled out just after the mysteries of life are revealed (in just two pages).

DFA Engine: Text-Directed

Contrast the regex-directed NFA engine with an engine that, while scanning the string, keeps track of all matches "currently in the works." In the tonight example, the engine knows a possible match has started the moment it hits `t`:

in string	in regex
after ... `ṭonight` ...	possible matches: `ṭo(nite|knight|night)`

Each subsequent character scanned updates the list of possible matches. After a few more characters are matched, the situation becomes

in string	in regex
after ... `toniġht` ...	possible matches: `to(niṭe|knight|niġht)`

with two possible matches in the works (and one alternative, knight, ruled out). Yet, with the `g` that follows, only the third alternative remains viable. Once the `h` and `t` are scanned as well, the engine realizes it has a complete match and can return success.

I call this "text-directed" matching because each character scanned controls the engine. As in the example above, a partial match might be the start of any number of different, yet possible, matches. Matches that are no longer viable are pruned as subsequent characters are scanned. There are even situations where a "partial match in progress" is also a full match. With ⌈`to(…)?`⌋, for example, if any match in the works ends inside the parentheses, a full match (of '`to`') is already confirmed and in reserve in case the longer matches don't pan out.

If you reach a character in the text that invalidates all the matches in the works, you must either: *1)* revert to one of the full matches in reserve, or, failing that, *2)* declare that there are no matches at the attempt's starting point.

Foreshadowing

If you compare these two engines based only on what I've mentioned so far, you might conclude that the text-directed DFA engine is generally faster. The regex-directed NFA engine might waste time attempting to match different subexpressions against the same text (such as the three alternatives in the example).

You would be right. During the course of an NFA match, the same character of the target might be checked by many different parts of the regex (or even by the same part, over and over). Even if a subexpression can match, it might have to be applied again (and again and again) as it works in concert with the rest of the regex to find a match. A local subexpression can fail or match, but you just never know about the overall match until you eventually work your way to the end of the regex. (You know, if I could find a way to include "It's not over until the fat lady sings." in this paragraph, I would.) On the other hand, a DFA engine is *determinate*—each character in the target is checked once (at most). When a character matches, you don't know yet if it will be part of the final match (it could be part of a possible match that doesn't pan out), but since the engine keeps track of all possible matches in parallel, it need be checked only once, period.

The Mysteries of Life Revealed

The foreshadowing in the last section might have been a bit thick, so I'd better come clean now, at least about some of it. The two basic technologies behind regular-expression engines have the somewhat imposing names *Nondeterministic Finite Automaton* (NFA) and *Deterministic Finite Automaton* (DFA). With mouthfuls like this, you see why I stick to just "NFA" and "DFA." We won't be seeing these phrases spelled out again.*

Because of the regex-directed nature of an NFA, the details of how the engine attempts a match are very important. As I said before, the writer can exercise a fair amount of control simply by changing how the regex is written. With the tonight example, perhaps less work would have been wasted had the regex been written differently, such as ⌈`to(ni(ght|te)|knight)`⌋, ⌈`tonite|toknight|tonight`⌋, or perhaps ⌈`to(k?night|nite)`⌋. With any given text, they all end up matching exactly the same text, but in doing so direct the engine in different ways. At this point, we don't know enough to judge which regexes, if any, are better than others, but that's coming soon.

It's the exact opposite with a DFA—since the engine keeps track of all matches simultaneously, none of these differences in representation matter so long as in the end they all represent the same possible matches. There could be a hundred different ways to achieve the same result, but since the DFA keeps track of them all simultaneously (almost magically—more on this later), it doesn't matter which form the regex takes. To a pure DFA, even expressions that appear as different as ⌈`abc`⌋ and ⌈`[aa-a](b|b{1}|b)c`⌋ are utterly indistinguishable.

It all boils down to. . .

Three things come to my mind when describing a DFA engine:

- DFA matching is very fast

- DFA matching is very consistent
- Talking about DFA matching is very boring

(I'll eventually expand on all these points.)

---

* I suppose I could explain the underlying theory that goes into these names . . . if I only knew it! As I hinted, the word *determinate* is pretty important, but for the most part the theory is not so long as we understand the practical effects. By the end of this chapter, we will. However, do see the sidebar on page 104.

The regex-directed nature of an NFA makes it interesting to talk about. NFAs provide plenty of room for creative juices to flow. There are great benefits in crafting an expression well, and even greater penalties for doing it poorly. A gasoline engine is not the only engine that can stall and conk out completely. To get to the bottom of this, we need to look at the essence of an NFA engine: *backtracking*.

Backtracking

The essence of an NFA engine is this: it considers each subexpression or component in turn, and whenever it needs to decide between two equally viable options, it selects one and remembers the other to return to later if need be. If the attempted option is successful and the rest of the regex is also successful, you are finished with the match. If anything in the rest of the regex eventually causes failure, the regex engine knows it can *backtrack* to where it chose the option and can continue with the match by trying another. This way, it eventually tries all possible permutations of the regex (or at least as many as needed until a match is found).

A Really Crummy Analogy

Backtracking is like leaving a pile of bread crumbs at every fork in the road. If the path you choose turns out to be a dead end, you can retrace your steps, giving up ground until you come across a pile of crumbs that indicates an untried path. Should that path, too, turn out to be a dead end, you can continue to backtrack, retracing your steps to the next pile of crumbs, and so on, until you eventually find a path that leads to your goal or until you run out of untried paths.

There are various situations when the regex engine needs to choose between two (or more) options—the alternation we saw earlier is only one example. Another example is that upon reaching⌈`...x?...`⌋, the engine must decide whether it should attempt ⌈`x`⌋ or not. Upon reaching ⌈`...x+...`⌋, however, there is no question about trying to match ⌈`x`⌋ at least once the plus requires at least one match, and that's nonnegotiable. Once the first ⌈`x`⌋ has been matched, though, the *requirement* is lifted and it then must decide to match another ⌈`x`⌋ or not. If it decides to match, it must decide if it will then attempt to match another. . . and another. . . and so on. At each of these many decision points, a virtual "pile of crumbs" is left behind as a reminder that another option (to match or not to match, whichever wasn't chosen at each point) remains viable at that point.

A crummy little example

Let's look at a full example using our earlier ⌈`to (nite|knight|night)`⌋ regex on the string '`hot▪tonic▪tonight!`' (silly, yes, but a good example). The first component ⌈`t`⌋ is attempted at the start of the string. It fails to match `h`, so the entire regex fails at that point. The engine's transmission then bumps along to retry the regex from the second position (which also fails), and again at the third. This time the ⌈t⌋ matches, but the subsequent ⌈•⌋ fails, so again the whole attempt fails.

The attempt that eventually starts at ...tonic... is more interesting. Once the to has been matched, the three alternatives become three available options. The regex engine picks one to try, remembering the others ("leaving some bread crumbs") in case the first fails. For the purposes of discussion, let's say that the engine first chooses ⌈nite⌋. That expression breaks down to "⌈n⌋ + ⌈i⌋ + ⌈t⌋ …," which gets to ...tonic... before failing. Unlike the earlier failures, this failure doesn't mean the e*nd* of the overall attempt because other options still remain. The engine chooses one, we'll say [knight, but it fails right away. That leaves only one final option, [night], but it eventually fails. Since that was the final untried option, its failure means the failure of the entire attempt starting at ...tonic..., so the transmission kicks in again.

Once the engine gets to the attempt starting at ...tonight! it gets interesting again, but this time, the ⌈night⌋ alternative successfully matches to the end. The successful matching to the end of the regex means an overall match, so the engine can report success at that point.

Two Important Points on Backtracking

The general idea of how backtracking works is fairly simple, but some of the details are quite important for real-world use. Specifically, when faced with multiple choices, which choice should be tried first? Secondly, when forced to backtrack, which saved choice should the engine use?

> In situations where the decision is between "make an attempt" and "skip an attempt," as with items governed by a question, star, and the like, the engine always chooses to first make the attempt. It will return later (to try skipping the item) *only if forced by the overall need to reach a global expression-wide match.*

This simple rule has far-reaching repercussions. For starters, it helps explain regex greediness, but not completely. To complete the picture, we need to know which (among possibly many) saved options to use when we backtrack. Simply put:

> The most recently saved option is the one returned to when a local failure forces backtracking. It's LIFO (last in first out).

This is easily understood in the crummy analogy—if your path becomes blocked, you simply retrace your steps until you come across a pile of bread crumbs. The first you'll return to is the most recently laid. The traditional analogy for describing LIFO also holds: like stacking and unstacking dishes, the most-recently stacked will be the first you'll unstack.

NFA: Theory vs. Reality

The true mathematical and computational meaning of "NFA" is different from what is commonly called an "NFA regex engine." In theory, NFA and DFA engines should match exactly the same text and have exactly the same features. In practice, the desire for richer, more expressive regular expressions has caused their semantics to diverge. We'll see several examples later in this chapter, but one right off the top is support for backreferences.

As a programmer, if you have a true (mathematically speaking) NFA regex engine, it is a relatively small task to add support for backreferences. A DFA's engine's design precludes the adding of this support, but an NFA's common implementation makes it trivial. In doing so, you create a more powerful tool, but you also make it decidedly *nonregular* (mathematically speaking). What does this mean? At most, that you should probably stop calling it an NFA, and start using the phrase "nonregular expressions," since that describes (mathematically speaking) the new situation. No one has actually done this, so the *name* "NFA" has lingered, even though the implementation is no longer (mathematically speaking) an NFA.

What does all this mean to you, as a user? Absolutely nothing. As a user, you don't care if it's regular, nonregular, unregular, irregular, or incontinent. So long as you know what you can expect from it (something this chapter will show you), you know all you need to care about.

For those wishing to learn more about the theory of regular expressions, the classic computer-science text is chapter 3 of Aho, Sethi, and Ullman's *Compilers—Principles, Techniques, and Tools* (Addison-Wesley, 1986), commonly called "The Dragon Book" due to the cover design. More specifically, this is the "red dragon." The "green dragon" is its predecessor, Aho and Ullman's *Principles of Compiler Design.*

Saved States

In NFA regular expression nomenclature, the piles of bread crumbs are known as saved *states.* A state indicates where a test can restart from if need be. It reflects both the position in the regex and the point in the string where an untried option begins. Because this is the basis for NFA matching, let me go over what I've already said with some simple but verbose examples. If you're comfortable with the discussion so far, feel free to skip ahead.

A match without backtracking

Let's look a simple example, matching ⌈`ab?c`⌋ against `abc`. Once the ⌈`a`⌋ has matched, the *current state* of the match is reflected by:

at 'abc'	matching ab?c

However, now that ⌜`b?`⌟ is up to match, the regex engine has a decision to make: attempt the ⌜`b`⌟ or not. Well, since `?` is greedy, it attempts the match. But so that it can recover if that attempt fails or eventually leads to failure, it adds

at 'abc'	matching ab?c

to its otherwise empty list of saved states. This indicates that the engine can later pick up the match in the regex just *after* the ⌜`b?`⌟, picking up in the text from just before the `b` (that is, where it is now). Thus, in effect, skipping the ⌜`b`⌟ as the question mark allows.

Once the engine carefully places that pile of crumbs, it goes ahead and checks the ⌜`b`⌟. With the example text, it matches, so the new current state becomes:

at 'abc'	matching ab?c

The final ⌜`c`⌟ matches as well, so we have an overall match. The one saved state is no longer needed, so it is simply forgotten.

A match after backtracking

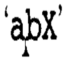
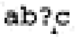
Now, if '`ac`' had been the text to match, everything would have been the same until the ⌜`b`⌟ attempt was made. Of course, this time it wouldn't match. This means that the path that resulted from actually attempting the ⌜`…?`⌟ failed. Since there is a saved state available to return to, this "local failure" does not mean overall failure. The engine backtracks, meaning that it takes the most recently saved state as its new current state. In this case, that would be the

at 'ac'	matching ab?c

that had been saved as the untried option before the ⌈b⌋ had been attempted. This time, the ⌈c⌋ and c match up, so the overall match is achieved.

A non-match

Now let's look at the same expression, but against abX. Before the ⌈b⌋ is attempted, the question mark causes the state

at 	matching 

to be saved. The ⌈b⌋ matches, but that avenue later turns out to be a dead end because the ⌈c⌋ fails to match X. The failure results in a backtrack to the saved state. The engine next tests ⌈c⌋ against the b that the backtrack effectively "unmatched." Obviously, this test fails, too. If there were other saved states, another backtrack would occur, but since there aren't any, the overall match at the current starting position is deemed a failure.

Are we done? No. The engine's transmission will still do its "bump along the string and retry the regex," which might be thought of as a pseudo-backtrack. The match restarts at:

at 'abX'	matching ab?c

The whole match is attempted again from the new spot, but like before, all paths lead to failure. After the next two attempts (from abX and abX) similarly fail, a true overall failure is finally reported.

Backtracking and Greediness

For tools that use this NFA regex-directed backtracking engine, understanding how backtracking works with your regular expression is the key to writing expressions that accomplish what you want, and accomplish it quickly. We've seen how ⌈`?`⌋ greediness works, so let's look at star (and plus) greediness.

Star, plus, and their backtracking

If you consider ⌈`x*`⌋ to be more or less the same as ⌈`x?x?x?x?x?x?…`⌋ (or, more appropriately, ⌈`(x(x(x(x…?)?)?)?)?)`⌋,* it's not too different from what we have already seen. Before checking the item quantified by the star, the engine saves a state indicating that if the check fails (or leads to failure), the match can pick up after the star. This is done over and over, until the star match actually does fail.

Thus, when matching ⌈`[0-9]+`⌋ against '`a•1234•num`', once ⌈`[0-9]`⌋ fails trying to match the space after the `4`, there are four saved states indicating that the match can pick up in the regex at `[0-9]+` at each of the string positions:

a 1▴234 num

a 12▴34 num

a 123▴4 num

a 1234▴ num

These represent the fact that the attempt of ⌜`[0-9]`⌟ had been optional at each of these positions. When it fails to match the space, the engine backtracks to the most recently saved state (the last one listed), picking up at 'a·1234▴·num' in the text and at `[0-9]+▴`, in the regex. Well, that's at the end of the regex. Now that we're actually there and notice it, we realize that we have an overall match.

Note that in the above list of four string positions, 'a·▴1234·num' is not a member because the first match using the plus quantifier is required, not optional. Would it have been in the list had the regex been ⌜`[0-9]*`⌟? *(hint: it's a trick question)*

❖ Turn the page to check your answer.

---

* Just for comparison, remember that a DFA doesn't care much about the form you use to express which matches are possible; the three examples *are* identical to a DFA.

Revisiting a fuller example

With our more detailed understanding, let's revisit the ⌈`^.*([0-9][0-9])`⌋ example from page 97. This time, instead of just pointing to "greediness" to explain why the match turns out as it does, we can use our knowledge of the mechanics of a match to explain why in precise terms. (If you're feeling a bit snowed in with the details, feel free to skim ahead for the general feel of what this chapter offers you can review the examples in more detail later on.)

I'll use '`CA•95472,•USA`' as an example. Once the ⌈`.*`⌋ has successfully matched to the end of the string, there are a dozen saved states accumulated from the star-governed dot matching 12 things that are (if need be) optional. These states note that the match can pick up in the regex at ⌈`^.*▲([0-9][0-9])`⌋ and in the string at each point where a state was created.

Now that we've reached the end of the string and pass control to the first ⌈`[0-9]`⌋, the match obviously fails. No problem: we have a saved state to try (a baker's dozen of them, actually). We backtrack, resetting the current state to the one most recently saved, to just before where ⌈`.*`⌋ matched the final `A`. Skipping that match (or "unmatching" it, if you like) leaves us trying that `A` against the first ⌈`[0-9]`⌋. It fails.

This backtrack-and-test cycle continues until the engine effectively unmatches the `2`, at which point the first ⌈`[0-9]`⌋ can match. The second, however, can't, so we must continue to backtrack. It's now irrelevant that the first ⌈`[0-9]`⌋ matched during the previous attempt—the backtrack resets the current state to before the first ⌈`[0-9]`⌋. As it turns out, the same backtrack resets the string position to just before the `7`, so the first ⌈`[0-9]`⌋ can match again. This time, so can the second (matching the 2). Thus, we have a match: '`CA•95472`,•`USA`', with `$1` getting `72`.

A few observations: first, the backtracking also entailed maintaining the status of the text being matched by the subexpression within parentheses. The backtracks always caused the match to be picked up at ⌜`^.*([0-9][0-9])`⌟ As far as the simple match attempt is concerned, this is the same as ⌜`^.*[0-9][0-9]`⌟ so I used phrases such as "picks up before the first ⌜`[0-9]`⌟." However, moving in and out of the parentheses involves updating the status of what `$1` should be, and this impacts efficiency. This is discussed fully in the next chapter (☞ 150).

It is important to realize that something governed by star (or any of the quantifiers) first matches as much as it can *without regard to what might follow in the regex.* In our example, the ⌜`.*`⌟ does not magically know to stop at the first digit, or the second to the last digit, or any other place *until the dot finally fails.* We saw this earlier when looking at how ⌜`^.*([0-9]+)`⌟ would never match more than a single digit by the ⌜`[0-9]+`⌟ part (☞ 98).

Where does ⌈`[0-9]*`⌋ match?

❖ *Answer to the question on page 106.*

No, `'a 1234 num'` would not be part of a saved state during a match of ⌈`[0-9]*`⌋. I posed this question because it's a mistake that new users commonly make.

Remember, a component that has star applied can *always* match. If that's the entire regex, the entire regex can always match. This certainly includes the attempt when the transmission applies the engine the first time, at the start of the string. In this case, the regex matches at `'a·1234·num'` and that's the end of it—it never even gets as far the digits.

More About Greediness

Many concerns (and benefits) of greediness are shared by both an NFA and a DFA. I'd like to look at some ramifications of greediness for both, but with examples explained in terms of an NFA. The lessons apply to a DFA just as well, but not for the same reasons. A DFA is greedy, period, and there's not much more to say after that. It's very constant to use, but pretty boring to talk about. An NFA, however, is interesting because of the creative outlet its regex-directed nature provides. An NFA engine affords the regex author direct control over how a match is carried out. This provides many benefits, as well as some efficiency-related pitfalls. (Discussions of efficiency are taken up in the next chapter.)

Despite these differences, the match results are often similar. For the next few pages, I'll talk of both engine types, but offer the description in the more easily understandable regex-directed terms of an NFA. By the end of this chapter, we'll have a firm grasp of just when the results might differ, as well as exactly why.

Problems of Greediness

As we saw with the last example, ⌈`.*`⌋ always marches to the end of the line.* This is because ⌈`.*`⌋ just thinks of itself and grabs what it can, only later giving up something if it is required to achieve an overall match.

Sometimes this can be a real pain. Consider a regex to match text wrapped in doublequotes. At first, you might want to write ⌈`".*"`⌋, but knowing what we know about ⌈`.*`⌋, guess where it will match in:

```
The name "McDonald's" is said "makudonarudo" in Japanese
```

---

* Or, with a tool where a dot can also match a newline, and strings that contain multi-line data, it matches through all the logical lines to the end of the string.

Actually, since we understand the mechanics of matching, we don't need to guess—we can *know*. Once the initial quote matches, ⌈`.*`⌋ is free to match, and immediately does so all the way to the end of the string. It will back off (or, perhaps more appropriately, *be backed off* by the regex engine) only as much as is needed until the final quote can match. Running this through in your head, you realize that it will match

`The name "McDonald's" is said "makudonarudo" in Japanese`

which is obviously not the doublequoted string that was intended. This is one reason why I caution against the overuse of ⌈`.*`⌋, as it can often lead to surprising results if you don't pay careful attention to greediness.

So, how can we have it match only `"McDonald's"`? The key is to realize that we don't want "anything" between the quotes, but rather "anything except a quote." If we use ⌈`[^"]*`⌋ rather than ⌈`.*`⌋ it won't overshoot the closing quote.

The regex engine's basic approach with ⌈`"[^"]*"`⌋ is exactly the same as before. Once the initial quote matches, ⌈`[^"]*`⌋ gets a shot at matching as much as it can. In this case, that's up to the quote after McDonald' s, at which point it finally stops because ⌈`[^"]`⌋ can't match the quote. So, at that point, control moves to the closing quote. It happily matches, resulting in overall success:

`The name "McDonald's" is said "makudonarudo" in Japanese`

Multi-Character "Quotes"

In the first chapter, I talked a bit about matching HTML tags, such as with the sequence …`<B>very</B>`… causing the "very" to be rendered in bold if the browser can do so. Attempting to match a `<B>`…`</B>` sequence seems similar to matching a quoted string, except the "quotes" in this case are the multi-character sequences `<B>` and `</B>`. Like the quoted string example, multiple sets of "quotes" cause problems:

…`<B>Billions</B> and <B>Zillions</B> of suns`

If we use ⌈`<B>.*</B>`⌋, the greedy ⌈`.*`⌋ causes the match-in-progress to zip to the end of the line, backtracking only far enough to allow the ⌈`</B>`⌋ to match, matching the last `</B>` on the line instead of the one corresponding to the opening ⌈`<B>`⌋ at the start of the match.

Unfortunately, since the closing delimiter is more than one character, we can't solve the problem in the same way as we did with doublequoted strings. We can't expect something as ridiculous as ⌈`<B>[^</B>]*</B>`⌋ to work. A character class represents only one character and not the full `</B>` sequence that we want.[*]

---

[*] Don't let the apparent structure of ⌈`[^</B>]`⌋ fool you. It is just a class to match one character, any character except `<`, `>`, `/`, and `B`. It is the same as, say ⌈`[^/<>B]`⌋, and certainly won't work as an "anything not `</B>`" construct.

Laziness?

This whole problem arises because star and friends (the quantifiers) are greedy. For a moment, let's imagine they are "lazy" (or "lax" or "minimal-matching" or "non-greedy" or "ungreedy" or whatever you'd like to call it). With a lazy ⌈`<B>.*</B>`⌋ and the

```
…<B>Billions</B> and <B>Zillions</B> of suns…
```

example, after the initial ⌈`<B>`⌋ has matched, a lazy ⌈`.*`⌋ would immediately decide that since it didn't require any matches, it would lazily not bother trying to perform any. So, it would immediately pass control to the following ⌈`<`⌋.

The ⌈`<`⌋ wouldn't match at that point, so control would return back to the lazy ⌈`.*`⌋, where it still had its untried option to attempt a match (to attempt multiple matches, actually). It would begrudgingly do so, with the dot then matching …`<B>Billions`… Again, the star has the option to match more, or to stop. We're assuming it's lazy for the example, so it first tries stopping. The subsequent ⌈`<`⌋ still fails, so ⌈`.*`⌋ has to again exercise its untried match option. After eight cycles, ⌈`.*`⌋ will have eventually matched Billions, at which point the subsequent ⌈`<`⌋ (and the whole ⌈`</B>`⌋ subexpression) will finally be able to match:

```
<B>Billions</B> and <B>Zillions</B> of suns
```

So, as we've seen, the greediness of star and friends can be a real boon at times, while troublesome at others. Having non-greedy, lazy versions is wonderful, as they allow you to do things that are otherwise very difficult (or even impossible). As it turns out, Perl provides ungreedy quantifiers in addition to the normal greedy versions. Like most great inventions, the idea is simple; we just had to wait for someone to think of it (in this case, Perl's author, Larry Wall).

Unfortunately, if you are not using Perl and don't have a lazy star quantifier. you are still faced with how to solve the `<B>…</B>` multi-character quote problem. Frankly, it is quite difficult to solve using a single, straight regular expression—I recommend splitting the work into two parts, one to find the opening delimiter, the other to search from that point to find the closing delimiter.

Greediness Always Favors a Match.

Recall the price-fixing (so to speak) example from Chapter 2 (☞ 46). Due to floating-point representation problems, values that should have been "1.625" or "3.00" were sometimes coming out like "1.62500000002828" and "3.00000000028822". To fix this, I used

```
     $price =~ s/(\.\d\d[1-9]?)\d*/$1/
```

to lop off all but the first two or three decimal digits from the value stored in the variable `$price`. The 「`\.\d\dd`」 matches the first two decimal digits regardless, while the 「`[1-9]?`」 matches the third digit only if it is non-zero.

I then noted:

> Anything matched so far is what we want to *keep*, so we wrap it in parentheses to capture to `$1`. We can then use `$1` in the replacement string. If this is the only thing that matches, we replace exactly what was matched with itself—not very useful. However, we go on to match other items outside the `$1` parentheses. They don't find their way to the replacement string, so the effect is that they're removed. In this case, the "to be removed" text is any extra digits, the ⌈`\d*`⌋ at the end of the regex.

So far so good, but let's consider what happens when the contents of the variable `$price` is already well-formed. When it is `27.625`, the ⌈`(\.\d\d[1-9]?)`⌋ part matches the entire decimal part. Since the trailing ⌈`\d*`⌋ doesn't match anything, the substitution replaces the '`.625`' with '.`625`'   an effective no-op.

This is the desired result, but wouldn't it be just a bit more efficient to do the replacement only when it would have some real effect? In such a case, the ⌈`\d*`⌋ would have to actually match at least one digit, since only it matches text to be omitted.

Well, we know how to write "at least one digit"! Simply replace ⌈`\d*`⌋ with ⌈`\d+`⌋:

```
$price =~ s/(\.\d\d[1-9]?)\d+/$1/
```

With crazy numbers like "1.62500000002828", it still works as before, but with something such as "9.43", the trailing ⌈`\d+`⌋ isn't able to match, so rightly, no substitution occurs. So, this is a great modification, yes? *No!* What happens with a three-digit decimal value like `27.625`?

Stop for a moment to work through the match of `27.625` yourself.

In hindsight, the problem is really fairly simple. Picking up in the action once ⌜`(\.\d\d[1-9]?)\d+`⌟ has matched `27.625`, we find that ⌜`\d+`⌟ can't match. That's no problem for the regex engine, since the match of '`5`' by ⌜`[1-9]`⌟, was *optional* and there is still a saved state to try. This is the option of having ⌜`[1-9]?`⌟ match nothing, leaving the `5` to fulfill the must-match-one requirement of ⌜`\d+`⌟. Thus, we get the match, but not the right match: `.625` is replaced by `.62`, and the value becomes incorrect.

The lesson here is that a match *always* takes precedence over a non-match, and this includes taking from what had been greedy if that's what is required to achieve a match.*

---

* A feature I think would be useful, but that no regex flavor that I know of has, is what I would call *possessive quantifiers*. They would act like normal quantifiers except that once they made a decision that met with local success, they would never backtrack to try the other option. The text they match could be unmatched if their enclosing subexpression *was* unmatched, but they would never give up matched text of their own volition, even in deference to an overall match. A possessive question mark would have solved the problem in ⌜ ⌟.

Is Alternation Greedy?

The only major control I haven't yet discussed in depth is ⌈`|`⌋, alternation. How alternation works is an important point because it can work in fundamentally different ways with different regex engines. When alternation is reached, any number of the alternatives might be able to match in the string, but which will? The Second Rule of Regular Expressions refers to quantifiers (star and friends) but not alternation. Is alternation greedy?

Let's look at an NFA engine. When faced with alternation, it tries each alternative in turn. You can pretty much count on each alternative being checked in the order given in the expression. Let's say the regex is ⌈`^(Subject|Date):␣`⌋. When the alternation is reached, the first alternative, ⌈`Subject`⌋, is attempted. If it matches, the rest of the regex, ⌈`:␣`⌋, is given a chance. If it turns out that it can't match, and if other alternatives remain (in this case, ⌈`Date`⌋), the regex engine will backtrack to try them. **This is just another case of the regex engine backtracking to a point where untried options are still available.** This continues until an overall match is achieved, or until all options (in this case, alternatives) are exhausted.

What text will actually be matched by ⌈`tour|to|tournament`⌋ when applied to the string '`three␣tournaments␣won`'? All the alternatives are attempted (and fail) during each attempt (at the 1st character position, 2nd, 3rd, and so on) until the transmission starts the attempt at `'three·tournaments·won'`. This time, the first alternative, tour, matches. Since the alternation is the last thing in the regex, the moment the tour matches, the whole regex is done. The other alternatives are not even tried again.

So, we see that alternation is *not* greedy, at least not for an NFA. Well, to be specific, alternation is not greedy for a *Traditional* NFA. Greedy alternation would have matched the longest possible alternative (⌈`tournament`⌋), wherever in the list it happened to be. A POSIX NFA, or any DFA would have indeed done just that, but I'm getting a bit ahead of myself.

To make sure you are on your toes, let me ask: which kind of alternation would result in ⌈`tour|to|tournament`⌋ matching the same text as ⌈`to(ur(nament)?)?`⌋? Before answering, make sure you realize that both are logically the same: they can match tour, to, and tournament, but nothing else. The question here is, in practice, which text will ⌈`to(ur(nament)?)?`⌋? actually match: `tour` (as with non-greedy alternation), `tournament` (as with greedy alternation), or something else altogether? ❖ Turn the page to check your answer.

Is Alternation Greedy?

The only major control I haven't yet discussed in depth is ⌈`|`⌋, alternation. How alternation works is an important point because it can work in fundamentally different ways with different regex engines. When alternation is reached, any number of the alternatives might be able to match in the string, but which will? The Second Rule of Regular Expressions refers to quantifiers (star and friends) but not alternation. Is alternation greedy?

Let's look at an NFA engine. When faced with alternation, it tries each alternative in turn. You can pretty much count on each alternative being checked in the order given in the expression. Let's say the regex is ⌈`^(Subject|Date):•`⌋. When the alternation is reached, the first alternative, ⌈`Subject`⌋, is attempted. If it matches, the rest of the regex, ⌈`:•`⌋, is given a chance. If it turns out that it can't match, and if other alternatives remain (in this case, ⌈`Date`⌋), the regex engine will backtrack to try them. **This is just another case of the regex engine backtracking to a point where untried options are still available.** This continues until an overall match is achieved, or until all options (in this case, alternatives) are exhausted.

What text will actually be matched by ⌈`tour|to|tournament`⌋ when applied to the string '`three•tournaments•won`'? All the alternatives are attempted (and fail) during each attempt (at the 1st character position, 2nd, 3rd, and so on) until the transmission starts the attempt at '`three·‸tournaments·won`'. This time, the first alternative, tour, matches. Since the alternation is the last thing in the regex, the moment the tour matches, the whole regex is done. The other alternatives are not even tried again.

So, we see that alternation is *not* greedy, at least not for an NFA. Well, to be specific, alternation is not greedy for a *Traditional* NFA. Greedy alternation would have matched the longest possible alternative (⌈`tournament`⌋), wherever in the list it happened to be. A POSIX NFA, or any DFA would have indeed done just that, but I'm getting a bit ahead of myself.

To make sure you are on your toes, let me ask: which kind of alternation would result in ⌈`tour|to|tournament`⌋ matching the same text as ⌈`to(ur(nament)?)?`⌋? Before answering, make sure you realize that both are logically the same: they can match tour, to, and tournament, but nothing else. The question here is, in practice, which text will ⌈`to(ur(nament)?)?`⌋? actually match: `tour` (as with non-greedy alternation), `tournament` (as with greedy alternation), or something else altogether? [5] Turn the page to check your answer.

Uses for Non-Greedy Alternation

Let's revisit the ⌈`(\.\d\d[1-9]?)\d*`⌋ example from page 110. If we realize that ⌈`\.\d\d[1-9]?`⌋, in effect, says "allow either ⌈ `\.\d\d]`⌋ or ⌈`\.\d\d[1-9]`⌋' we can rewrite the entire expression as ⌈`(\.\d\d|\.\d\d[1-9])\d*`⌋. (There is not a compelling reason to make this change it's merely a handy example.) Is it *really* the same as ⌈`(\.\d\d[1-9]?)\d*`⌋? If alternation is greedy, then yes, it is, but the two are quite different with non-greedy alternation.

Let's consider it as non-greedy for the moment. If the first alternative, ⌈`\.\d\d`⌋, is able to match, the ⌈`\d*`⌋ that follows the alternation will certainly succeed. This might include matching a non-zero digit (which, if you'll recall the original problem, is a digit we really want to match only within the parentheses). Also, realize that the second alternative begins with a copy of the entire first alternative—if the first alternative fails, the second will certainly fail as well. The regex engine will nevertheless make the attempt, but I'll leave that issue of efficiency to the next chapter.

Interestingly, if we use ⌈`(\.\d\d[1-9]|\.\d\d)\d*`⌋, which is the same except that the alternatives have been swapped, we do effectively get a replica of the original ⌈`(\.\d\d[1-9]?)\d*`⌋. The alternation has meaning in this case because if the first alternative fails due to the ⌈`[1-9]`⌋, the second alternative still stands a chance.

In distributing the ⌈`[1-9]?`⌋ to two alternatives and placing the shorter one first, we fashioned a non-greedy ⌈`?`⌋ of sorts. It ends up being meaningless in this particular example because there is nothing that could ever allow the second alternative to match if the first fails. I see this kind of faux-alternation often, and it is invariably a mistake. In one book I've read, ⌈`a*((ab)*|b*)`⌋ is used as an example in explaining something about regex parentheses. A rather silly example, isn't it? Since the first alternative, ⌈`(ab)*`⌋, can never fail, any other alternatives (just ⌈`b*`⌋ in this case) are utterly meaningless. You could add

⌜`a*((ab)*|b*|.*|partridge•in•a•pear•tree|[a-z])`⌟

and it wouldn't change the meaning a bit.

Non-greedy alternation pitfalls

You can often use non-greedy alternation to your advantage by crafting just the match you want, but non-greedy alternation can also lead to unexpected pitfalls for the unaware. Consider wanting to match a January date of the form '`Jan 31`'. We need something more sophisticated than, say, ⌜`Jan•0123][0-9]`⌟, as that allows "dates" such as '`Jan 00`', '`Jan 39`', and disallows, say, '`Jan 7`'.

Is ⌈`to(ur(nament)?)?`⌋ Greedy?

❖ Answer to the question on page 112.

Once the initial ⌈`to`⌋ has matched, we know that an overall match is guaranteed since nothing in the regex that follows is required. Although this may well end up being the final overall match, the engine can't decide that yet, as there is still the possibility that more can be matched—the question marks are greedy, so they attempt to match what they quantify.

The ⌈`(ur(nament)?)`⌋ quantified by the outermost question mark matches if possible, and within that, the ⌈`nament`⌋ also matches if possible. So overall, the entire ⌈`to`⌋ + ⌈`ur`⌋ + ⌈`nament`⌋ matches if at all possible. In practice, it matches the same text as a greedy-alternation ⌈`tour|to|tournament`⌋: the longest possible.

One simple way to match the date part is to attack it in sections. To match from the first through the ninth, using ⌈`0?[1-9]`⌋ allows a leading zero. Adding ⌈`[12][0-9]`⌋ allows for the tenth through the 29th, and ⌈`3[01]`⌈ rounds it out. Putting it all together, we get ⌈`Jan•(0?[1-9]|[12][0-9]|3[01])`⌋.

Where do you think this matches in '`Jan 31 is my dad's birthday`'? We want it to match '`Jan 31`', of course, but non-greedy alternation actually matches only '`Jan 3`'. Surprised? During the match of the first alternative, ⌈`0?[1-9]`⌋, the leading ⌈`0?`⌋ fails, but the alternative matches because the subsequent ⌈`[1-9]`⌋ has no trouble matching the `3`. Since that's the end of the expression, the match is complete.

Were the order of the alternatives reversed, or the alternation greedy, this problem would not have surfaced. Another approach to our date-matching task could be ⌈`Jan•(31|[123]0|[012]?[1-9])`⌋. Like the first solution, this requires careful arrangement of the alternatives to avoid the problem. Yet a third approach is ⌈`Jan•(0[1-9]|[12][0-9]?|3[01]?|[4-9])`⌋, which works properly regardless of the ordering. Comparing and contrasting these three expressions can prove quite interesting (an exercise I'll leave for your free time, although the "A Few Ways to Slice and Dice a Date" sidebar on the next page should be helpful).

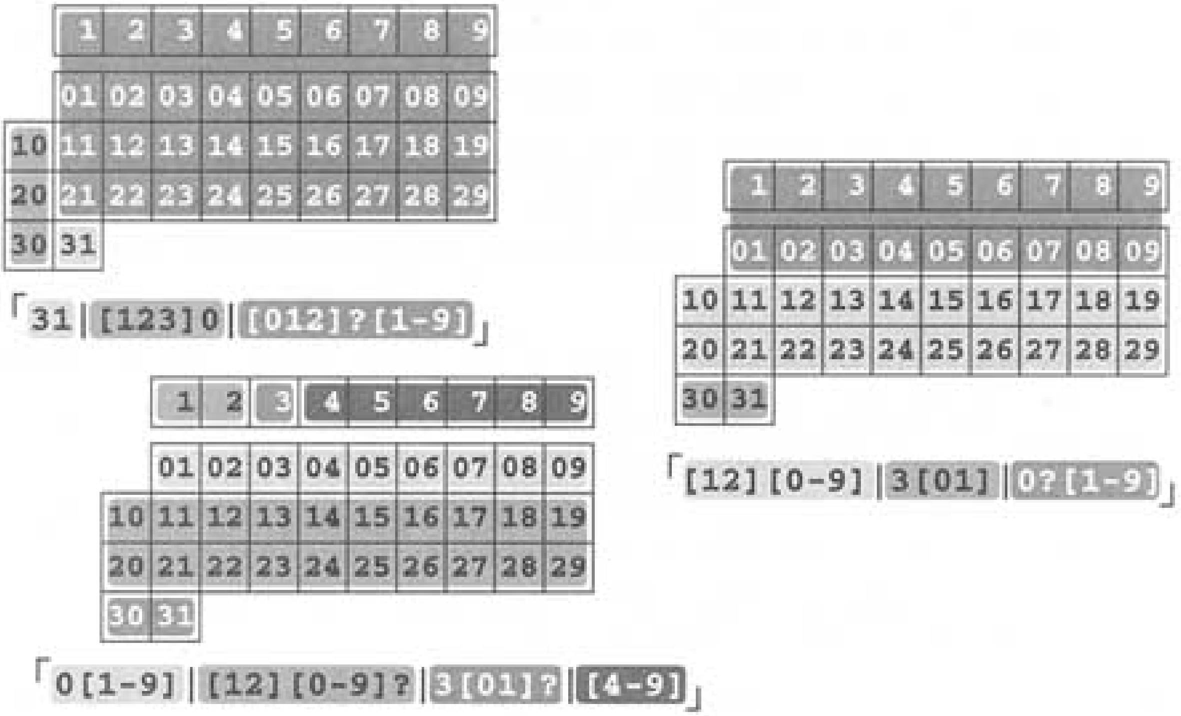
Greedy Alternation in Perspective

As we've seen, non-greedy alternation is more powerful than greedy alternation because you have more control over just how a match is attempted—it doesn't say "Take any of these" so much as "Try this, then that, and finally the other."

With an NFA, alternation can entail a lot of backtracking, and finding ways to reduce it is usually a good way to make the regex more efficient, which means faster execution. We'll see some examples soon, and more in the next chapter.

A Few Ways to Slice and Dice a Date

A few approaches to the date-matching problem posed on page 114. The calendar associated with each regex has what can be matched by alternative color-coded with the regex.



Character Classes vs. Alternation

Because of the superficial similarity between ⌈`[abc]`⌋ and ⌈`a|b|c`⌋, you might tend to think that character classes are implemented similarly, but with an NFA nothing could be further from the truth. With a DFA, it makes no difference one way or the other, but with an NFA a character class is an atomic unit which acts like a sieve, allowing a character to pass only if it is one of the target characters. This test is, in effect, done in parallel. It is much more efficient than the comparable NFA alternation, which checks each alternative in turn (entailing a lot of backtracking).

NFA, DFA, and POSIX

"The Longest-Leftmost"

Let me repeat: when the transmission starts a DFA engine from some particular point in the string, if any match is to be found from that position, the DFA will find the longest possible, period. Since it's the longest from among all possible matches that start equally furthest to the left, it's the "longest-leftmost" match.

Really, the longest

Issues of which match is longest aren't confined to alternation. Consider how an NFA matches the (horribly contrived) ⌜`one(self)?(selfsufficient)?`⌟ against the string `oneselfsufficient`. An NFA first matches ⌜`one`⌟ and then the greedy ⌜`(self)?`⌟, leaving ⌜`(selfsufficient)?`⌟ left to try against `sufficient`. That doesn't match, but that's okay since it is optional. So, the Traditional NFA returns `oneselfsufficient` and discards the untried states. (A POSIX NFA, on the other hand, is another story that we'll get to shortly.)

On the other hand, a DFA finds the longer `oneselfsufficient`. If the ⌜`(self)?`⌟ were to be non-greedy and go unmatched, the ⌜`(selfsufficient)?`⌟ would be able to match, yielding a longer overall match. That is the longest possible, so is the one that a DFA finds.

I chose this silly example because it's easy to talk about, but I want you to realize that this issue is important in real life. For example, consider trying to match *continuation lines*. It's not uncommon for a data specification to allow one logical line to extend across multiple real lines if the real lines end with a backslash before the newline. As an example, consider the following:*

```
    SRC=array.c builtin.c eval.c field.c gawkmisc.c io.c main.c
\
             missing.c msg.c node.c re.c version.c
```

You might normally want to use ⌜`^\w+=.*`⌟ to match this kind of "var = value" assignment line, but this regex doesn't consider the continuation lines. I'm assuming, for the example, that the tool's dot won't match a newline-you could substitute it with something like ⌜`[^\n]`⌟ if need be.

To match continuation lines, you might want to try appending ⌈`(\\\n.*)*`⌋ to the regex. Ostensibly, this says that any number of additional logical lines are allowed so long as they follow an escaped newline. **This seems reasonable, but it will never work with a traditional NFA**. By the time the original ⌈`.*`⌋ has reached the newline, *it has already passed the backslash*, and nothing in what was added forces it to backtrack. Yet a DFA would find the longer multi-line match if it existed, simply because it was, indeed, the longest.

POSIX and the Longest-Leftmost Rule

The POSIX standard requires that if you have multiple possible matches that start at the same position, the one matching the most text must be the one returned. Period.

---

* The actual text is irrelevant to the example, but for what it's worth, this is from the GNU *awk* makefile.

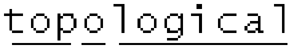
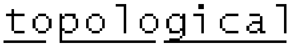
The standard uses the phrase "longest of the leftmost."* It doesn't say you have to use a DFA, so if you want to use an NFA when creating a tool, what's a programmer to do? If you want to implement a POSIX NFA, you'd have to find the full `oneselfsufficient` and all the continuation lines, despite these results being "unnatural" to your NFA.

A Traditional NFA engine stops with the first match it finds, but what if it were to continue to try all the remaining options? Each time it reached the end of the regex, it would have another plausible match. By the time *all* options are exhausted, it could simply report the longest of the plausible matches it had found. Thus, a POSIX NFA.

An NFA applied to the first example would, after matching ⌈`(self)?`⌋, have saved an option noting that it could pick up matching ⌈`one(self)?(selfsufficient)?`⌋ at `oneselfsufficient`. Even after finding the `oneselfsufficient` that a Traditional NFA returns, a POSIX NFA would continue to exhaustively check the remaining options, eventually realizing that yes, there was a way to match the longer `oneselfsufficient`.

Really, the leftmost

Not only does POSIX mandate that the longest-leftmost match be found, but that if subexpression capturing (via parentheses) is supported, each captured subexpression must capture the maximum amount of text *consistent with the overall leftmost-longest, and with its place in the regex.* This means that the overall match is chosen based only on the longest-leftmost rule, but once chosen, the first set of capturing parentheses gets the maximum possible from that. After that, the second set gets the maximum possible of what's left. And so on.

⌜`(to|top)(o|polo)?(gical|o?logical)`⌟ can match '`topological`' in various ways, but a POSIX engine would have to match  as shown (the part matched by each parenthesized expression is marked). Compare this to, say the  that a Traditional NFA would find. The former's first parentheses' `top` is longer than the latter's `to`, so it would have to be the particular match chosen for a POSIX match. Similarly, even though it could then match nothingness in the second set of parentheses (matching `ological` in the third), it takes the longest match that is consistent with both the longest overall match and the earlier parentheses taking their longest match.

Note, though, that many POSIX engines do not support any kind of capturing, so this issue is normally not a concern.

---

* The rationale associated with the standard uses the phrase "leftmost-longest," which is incorrect English considering what we know they're trying to say. It means "of all the equally 'longest choose the leftmost," which is quite different from "longest of the leftmost."

Speed and Efficiency

If efficiency is an issue with a Traditional NFA (and with all that backtracking, believe me, it is), it is doubly so with a POSIX NFA since there can be so much more backtracking. A POSIX NFA engine really does have to try every possible permutation of the regex every time. Examples in the next chapter show that poorly written regexes can suffer extremely severe performance penalties.

DFA efficiency

The text-directed DFA is a really fantastic way around all the inefficiency of backtracking. It gets its matching speed by keeping track of all possible ongoing matches at once. How does it achieve this magic?

The DFA engine spends extra time and memory before a match attempt to analyze the regular expression more thoroughly (and in a different way) than an NFA. Once it starts actually looking at the string, it has an internal map describing "If I read such-and-such a character now, it will be part of this-and-that possible match." As each character of the string is checked, the engine simply follows the map.

Building that map (called *compiling the regex*, something that must be done for an NFA as well, but it's not nearly as complex) can sometimes take a fair amount of time and memory, but once done for any particular regular expression, the results can be applied to an unlimited amount of text. It's sort of like charging the batteries of your electric car. First, your car sits in the garage for a while, plugged into the wall like a Dust Buster, but when you actually use it, you get consistent, clean power.

DFA and NFA in Comparison

Both DFA and NFA engines have their good and bad points:

Differences in the pre-use compile

Before applying a regex to a search, both types of engines compile the regex to an internal form suited to their respective matching algorithms. An NFA compile is generally faster, and requires less memory. There's no real difference between a Traditional and POSIX NFA compile.

Differences in match speed

For simple literal-match tests in "normal" situations, both types match at about the same rate. A DFA's matching speed is unrelated to the particular regex, while an NFA's is directly related. For a Traditional NFA to conclude that there is no match, it must try every possible permutation of the regex. This is why I spend the entire next chapter on techniques to write an NFA regex that will match quickly. As we'll see, an NFA match can sometimes take forever (well, almost). At least a Traditional NFA can stop if and when it finds a match. A POSIX NFA, on the other hand, must always try every possible permutation to ensure that it has found the longest possible match, so it generally takes the same (possibly very long) amount of time to complete a successful match as it does completing a failed attempt. Writing efficient regexes is doubly important for a POSIX NFA.

In one sense, I speak a bit too strongly in the previous paragraph, since optimizations can often reduce the work needed to return an answer. We've already seen the optimization of not trying ⌈^⌋ anchored regexes beyond the start of the string (☞ 92), and we'll see many more in the next chapter. In general, the need for optimizations is less pressing with a DFA (since its matching is so fast to begin with), but for the most part, the extra work done during the DFA's pre-use compile affords better optimizations than most NFA engines take the trouble to do.

Modern DFA engines often try to reduce the time and memory used during the compile by postponing some work until a match is attempted. Often, much of the compile-time work goes unused because of the nature of the text actually checked. A fair amount of time and memory can sometimes be saved by postponing the work until it's actually needed during the match. (The technobabble term for this is *lazy evaluation*.) It does, however, create cases where there can be a relationship between the regex and the match speed.

Differences in what is matched

A DFA (or anything POSIX) finds the longest leftmost match. A Traditional NFA might also, or it might find something else. Any individual engine will always treat the same regex/text combination in the same way, so in that sense it's not "random," but other NFA engines may decide to do slightly different things. Virtually all NFA engines I've seen work exactly the way I've described here,* but it's not something absolutely guaranteed by technology or any standard.

---

[*] I have seen two tools employ slightly different engines. Older versions of GNU *awk* (gawk), such as version 2.15.6, had neither greedy nor non-greedy alternation it seemed rather random what alternative would match. The other is MIFES, a popular Japanese editor. Some versions sometimes turn 「`.*x`」 into 「`[^x]*x`」 (in an effort, I suppose, to make regexes seem more "natural" to those that don't understand them).

Differences in capabilities

An NFA engine can support many things that a DFA cannot. Among them are:

- Capturing text matched by a parenthesized subexpression. Related features are backreferences and after-match information saying *where* in the text each parenthesized subexpression matched.

- Lookahead. Although we haven't it discussed in this chapter (because only Perl supports it:[*] ☞ 228), *positive lookahead* allows you to, in effect, say "This subexpression must match for me to continue on, but just check it, don't 'consume' any of the text." *Negative lookahead* is the analogous "this subexpression mustn't match."

- **[Traditional NFA only]** Non-greedy quantifiers and alternation. A DFA could easily support a guaranteed shortest overall match (although for whatever reason, this option never seems to be made available to the user), but it cannot implement the local laziness that we've talked about.

Differences in implementation ease

Although they have limitations, simple versions of DFA and NFA engines are easy enough to understand and to implement. The desire for efficiency (both in time and memory) and enhanced features drives the implementation to greater and greater complexity. With code length as a metric, consider that the NFA regex support in the Version 7 (January 1979) edition of *ed* was less than 350 lines of C code. (For that matter, the *entire* source for *grep* was a scant 478 lines.) Henry Spencer's 1986 freely available implementation of the Version 8 regex routines was almost 1,900 lines of C, and Tom Lord's 1992 POSIX NFA package rx (used in GNU *sed*, among other tools) is a stunning 9,700 lines long. For DFA implementations, the Version 7 *egrep* regex engine was a bit over 400 lines long, while Henry Spencer's 1992 full-featured POSIX DFA package is over 4,500 lines long. To provide the best of both worlds, GNU *egrep* Version 2.0 utilizes two fully functional engines (about 8,300 lines of code).

Simple, however, does not necessarily mean "lack of features." I recently wanted to use regular expressions for some text processing with Delphi, Borland's Pascal development environment. I hadn't used Pascal since college, but it still didn't take long to write a simple NFA regex engine. It doesn't have a lot of bells and whistles, and it is not built for maximum speed, but the flavor is relatively full-featured and so even the simple package is quite usable.

---

*.*lex* has *trailing context*, which is exactly the same thing as zero-width positive lookahead at the end of the regex, but it can't be generalized to embedded use.

DFA Speed With NFA Capabilities: Regex Nirvana?

I've said several times that a DFA can't provide capturing parentheses or backreferences. This is quite true, but it certainly doesn't preclude hybrid approaches which mix technologies in an attempt to reach regex nirvana. The sidebar on page 104 told how NFAs have diverged from the theoretical straight and narrow in search of more power, and it's only natural that the same happens with DFAs. A DFAs construction makes it more difficult, but that doesn't mean impossible.

GNU grep takes a simple but effective approach. It uses a DFA when possible, reverting to an NFA when backreferences are used. GNU awk does something similar—it uses GNU grep's fast shortest-leftmost DFA engine for simple "does it match" checks, and reverts to a different engine for checks where the actual extent of the match must be known. Since that other engine is an NFA, GNU awk can conveniently offer capturing parentheses, and it does via its special gensub function.

Until recently, there seemed to be little practical work done on extending the DFA itself, but there is active research in this area. As I finish up work on this book, Henry Spencer writes that his recent and mostly DFA package supports capturing parentheses, and is "at worst quadratic in text size, while an NFA is exponential." This new technology needs more time to mature before it becomes widely available, but it holds promise for the future.

Practical Regex Techniques

Now that we've touched upon the basic concerns for writing regular expressions, I'd like to put this understanding to work and move on to more advanced techniques for constructing regular expressions. I'll try to pick up the pace a bit, but be aware that some of the issues are still fairly complex.

Contributing Factors

Writing a good regex involves striking a balance among several concerns:

- matching what you want, but only what you want
- keeping the regex manageable and understandable
- for an NFA, being efficient (creating a regex that leads the engine quickly to a match or a non-match, as the case may be)

These concerns are often context-dependent. If I'm working on the command line and just want to *grep* something quickly, I'll probably not care if I match a bit more than I need, and I won't usually be too concerned to craft just the right regex for it. I'll allow myself to be sloppy in the interest of time, since I can quickly peruse the output for what I want. However, when I'm working on an important script, it's worth the time and effort to get it right: a complex regular expression is okay if that's what it takes. There is a balance among all these issues.

Even in a script, efficiency is also context-dependent. For example, with an NFA, something long like ⌈`^-(display|geometry|cemap|…|quick24|random|raw)$`⌋ to check command-line arguments is inefficient because of all that alternation, but since it is only checking command-line arguments (something done perhaps a few times at the start of the program) it wouldn't matter if it took 100 times longer than needed. It's just not an important place to worry much about efficiency. Were it used to check each line of a potentially large file, the inefficiency would penalize you for the duration of the program.

Be Specific

Continuing with the continuation-line example from page 116, we found that ⌈`^\w+=.*(\\\n.*)*`⌋ applied with a Traditional NFA wouldn't match both lines of:

```
     SRC=array.c builtin.c eval.c field.c gawkmisc.c io.c main.c
\
             missing.c msg.c node.c re.c version.c
```

The problem is that the first ⌈`.*`⌋, matches past the backslash, pulling it out from under the ⌈`(\\\n.*)*`⌋ that we want it to be matched by. If we don't want to match past the backslash, we should say that in the regex:*

⌈`^\w+=[^\n\\]*(\\\n[^\n\\]*)*`⌋

This might be too specific, though, since it doesn't allow backslashes except those at the end of lines. You can take advantage of a regex-directed NFA's non-greedy alternation, starting over with the original expression and changing the ⌈`[^\n\\]*`⌋ to ⌈`(\\\n|.)*`⌋, which gives us:

⌈`^\w+=(\\\n|.)*`⌋

The part we appended earlier, added to match the escaped newlines and their subsequent continuation lines, is now unneeded—the main ⌈`(\\\n|.)*`⌋ matches right through newlines, but only if they are escaped.

Well, not exactly. A line ending with `\\` (not common, but possible) happens to have a backslash before the newline, but the newline is *not* escaped so there is no continuation line. The problem is that the dot will match the first backslash, allowing the second to match ⌜`\\\n`⌟ on the next cycle of the star, resulting in a false continuation line. We weren't specific enough—if we want that the second alternative should not match an escape, we should say that using the ⌜`[^\n\\]`⌟ from before.

---

* Notice how I made sure to include `\n` in the class? You'll remember that one of the assumptions of the original regex was that dot didn't match a newline, and we don't want its replacement to match a newline either (☞ 79).

The problem with ⌜`(\\\n|[^\n\\])*`⌟ is that it now doesn't allow anything to be escaped except for newlines. We really want the first alternative to say "any escaped byte." If dot doesn't match newline, don't make the silly mistake of trying ⌜`[\n.]`⌟. If octal escapes are supported, ⌜`(\\[\000-\377]|[^\n\\])*`⌟ works.

Finally, a hint to what the next chapter covers: Since there's now no ambiguity between the two alternatives, we may as well swap them so that the one likely to be used most often comes first. This will allow an NFA to match a bit more quickly.

Matching an IP address

As another example that we'll take much further, let's match an IP (Internet Protocol) address: four numbers separated by periods, such as `1.2.3.4.` Often, the numbers are padded to three digits, as in `001.002.003.004`. If you want to check a string for one of these, you could use ⌜`[0-9]*\.[0-9]*\.[0-9]*\[0-9]*`⌟, but that's so vague that it even matches '`and then.....?`'. Look at the regex: it doesn't even *require* any numbers to exist—its only requirements are three periods (with nothing but digits, *if anything*, between).

To fix this regex, we first change the star to a plus, since we know that each number must have at least one digit. To ensure that the entire string is only the IP address, we wrap the regex with ⌜`^...$`⌟. This gives us:

⌜`^[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+$`⌟

Using Perl's ⌜`\d`⌟ as a shorthand for ⌜`[0-9]`⌟, this becomes the more-readable* ⌜`^\d+\.\d+\.\d+\.\d+$`⌟, but it still allows things that aren't IP addresses, like ⌜`1234.5678.9101112.131415`⌟. (Each number must be in the range of 0-255.) To enforce that each number is three digits long, you could use

⌜`^\d\d\d\.\d\d\d\.\d\d\d\.\d\d\d$`⌟

but now we are *too* specific. We still need to allow one- and two-digit numbers (as in `1.2.3.4`). If the tool's regex flavor provides the {*`min, max`*} notation, you can use ⌈`^\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}$`⌋, and if not, you can always use ⌈`\d\d?\d?`⌋ or ⌈`\d(\d\d?)?`⌋ for each part. Either regex allows one to three digits, but in a slightly different way.

Depending on your needs, you might be happy with some of the various degrees of vagueness in the expressions so far (just as my editor was comfortable with the simple expression he used on page 5). If you really want to be strict, you have to worry that ⌈`\d\d\d`⌋ can match `999`, which is above `255`, and thus an invalid component of an IP address.

---

* Or maybe not—it depends on what you are used to. If you are new to regular expressions, at first they all seem odd. Perl has perhaps the richest regular expression flavor to be found, and lacking enough symbols to serve as metacharacters, many items such as ⌈`\d`⌋ combine a backslash and a letter for their representation. To some, these added "features" are merely superficial gimmicks that add more backslashes. Personally, I don't like a lot of backslashes either, but I enjoy the features (superficial or not) and so use them.

Several approaches would ensure that only numbers from 0 to 255 appear. One silly approach is ⌈`0|1|2|3…|253|254|255`⌋. Actually, this doesn't allow the zero-padding that is allowed, so you really need ⌈`0|00|000|1|01|001|…`, which is even more ridiculous. For a DFA engine, it is ridiculous only in that it's so long and verbose—it still matches just as fast as any regex describing the same text. For an NFA, however, all the alternation kills efficiency.

A realistic approach would concentrate on which digits are allowed in a number, and where. If a number is only one or two digits long, there is no worry as to whether the value is within range, so ⌈`\d|\d\d`⌋ takes care of them. There's also no worry about the value for a three-digit number beginning with a `0` or `1`, since such a number is in the range `000-199`. This lets us add ⌈`[01]\d\d`⌋, leaving us with ⌈`\d|\d\d|[01]\d\d`⌋. You might recognize this as being similar to the date example earlier in this chapter (☞ 115), and the time example in Chapter 1 (☞ 24).

Continuing, a three-digit number beginning with a `2` is allowed if the number is 255 or less, so a second digit less than `5` means the number is valid. If the second digit *is* `5`, the third must be less than `6`. This can all be expressed as ⌈`2[0-4]\d|25[0-5]`⌋.

This may seem confusing at first, but the approach makes sense when you think about it. The result is ⌜`\d|\d\d|[01]\d\d|2[0-4]\d|25[0-5]`⌟. Actually, we can combine the first three alternatives to yield ⌜`[01]?\d\d?|2[0-4]\d|25[0-5]`⌟. Doing so is more efficient for an NFA, since any alternative that fails results in a backtrack. Note that using ⌜`\d\d?`⌟ in the first alternative, rather than ⌜`\d?\d`⌟, allows an NFA to fail just a bit more quickly when there is no digit at all. I'll leave the analysis to you—walking through a simple test case with both should illustrate the difference. We could do other things to make this part of the expression more efficient, but I'll leave that facet of the discussion for the next chapter.

Now that we have a subexpression to match a single number from 0 through 255, we can wrap it in parentheses and insert it in place of each ⌜`\d{1,3}`⌟ in the earlier regex. This gives us (broken across lines to fit the width of the page):

```
⌜^([01]?\d\d?|2[0-4]\d|25[0-5])\.([01]?\d\d?|2[0-4]\d|25[0-5])\.

([01]?\d\d?|2[0-4]\d|25[0-5])\.([01]?\d\d?|2[0-4]\d|25[0-5])$⌟
```

Quite a mouthful! Was the trouble worth it? You have to decide yourself based upon your own needs. This still allows `0.0.0.0`, which is invalid because all the digits are zero, but creating a regex to disallow this would be much tougher. You can't just disallow `0` from each number, since something like `123.202.0.188` is valid. At some point, depending on your needs, you have to decide when it is not worth the trouble to be more specific—the cost/benefit ratio starts to suffer from diminishing returns. Sometimes it's better to take some of the work out of the regex. For example, going back to ⌈`^\d{1,3}\.\d{1,3}\.\d{1,3\.\d{1,3}$`⌋ and wrapping each component in parentheses will stuff the numbers into `$1`, `$2`, `$3`, and `$4`, which can then be validated by other programming constructs.

One thing to consider, though, is the *negative lookahead* that Perl provides. It can disallow specific cases before the engine even tries the "main" expression. In this case, prepending ⌈`(?!0+\.0+\.0+\.0+$)`⌋ causes immediate failure when every component is zero. This is explained further in Chapter 7 (☞ 230).

Know your context

It's important to realize that the two anchors are required to make this regex work. Without them, it would be more than happy to match `ip=72123.3.21.993`, or for a Traditional NFA, even `ip=123.3.21.223`.

In that second case, it does not even fully match the final `223` that should have been allowed. Well, it is *allowed,* but there's nothing (such as a separating period, or the trailing anchor) to force that match. The final group's first alternative, ⌈`[01]?\d\d?`⌋, matched the first two digits, and then that was the end of the regex. Like with the date-matching problem on page 114, we can arrange the order of the alternatives to achieve the desired effect. In this case, that would be such that the alternatives matching three digits come first, so any proper three-digit number is matched in full before the two-digit-okay alternative is given a chance.

Rearranged or not, the first mistaken match is still a problem. "Ah!" you might think, "I can use word boundary anchors to solve this problem." Probably not. Such a regex could still match `1.2.3.4.5.6`. To disallow embedded matches, you must ensure the surrounding context, and word boundaries are not enough. Wrapping the entire regex in ⌈`(^|•)...(•|$)`⌋ is one idea, but what constitutes a "good solution" depends on the situation.

Difficulties and Impossibilities

Pinpointing what to specify in a regex can sometimes be difficult when you want almost anything, as we saw with the ⌈`".*"`⌋ example. Actually, in that example, we didn't really want "anything," but rather "anything except a doublequote," and so it was best to say that: ⌈`"[^"]*"`⌋.

Unfortunately, sometimes you can't express things so clearly. For example, if you want to allow escaped quotes within the text, such as with `"he•is•6'4\"•tall"`, the ⌈`[^"]*`⌋ would never be allowed past the escaped (nor any other) doublequote, and you could match too little (only `"he•is•6'4\"`•`tall"` in this case). A larger problem is when what you *don't* want is more than a single character, such as with the `<B>…</B>` example from page 109. We'll look at these kinds of problems and their solutions in the next section.

Matching balanced sets of parentheses, brackets, and the like presents another difficulty. Wanting to match balanced parentheses is quite common when parsing many kinds of configuration files, programs, and such. Take, for example, wanting to do some processing on all of a function's arguments when parsing a language like C. Function arguments are wrapped in parentheses following the function name, and may themselves contain parentheses resulting from nested function calls or generic math grouping. At first, ignoring that they may be nested, you might be tempted to use:

⌈`\bfoo\([^)]*\)`⌋

In hallowed C tradition, I use `foo` as the example function name. The marked part of the expression is ostensibly meant to match the function's arguments. With examples such as `foo(2,•4.0)` and `foo(somevar,•3.7)`, it works as expected. Unfortunately, it also matches `foo(bar(somevar),•3.7)`, which is not as we want. This calls for something a bit "smarter" than ⌈`[^)]*`⌋.

To match the parenthesized expression part you might consider the following regular expressions (among others):

1.	`\(.*\)`	literal parentheses with anything in between
2.	`\([^)]*\)`	from an opening parenthesis to the next closing parenthesis
3.	`\([^()]*\)`	from an opening parenthesis to the next closing parenthesis, but no other opening parentheses allowed in between

Figure 4-1 illustrates where these match against a sample line of code.

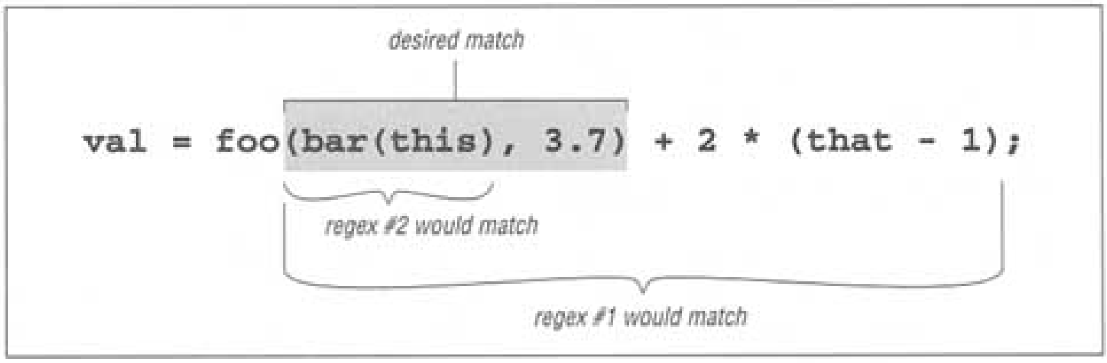


Figure 4-1:
Match locations of our sample regexes

We see that regex #1 matches too much,* and regex #2 matches too little. Regex #3 doesn't even match successfully—in isolation it would match `'(this)'`, but because it must come immediately after the foo, it fails. So, none of these expressions works. In fact, the problem is that *you simply can't match arbitrarily nested constructs with regular expressions.* It just can't be done.

You can construct a regular expression that matches nested constructs up to a *certain depth,* but not to any arbitrary level of nesting. A regular expression to allow just one level of nesting is the monstrosity ⌈`\([^()]*(\([^()]*\)[^()]*)*\)`⌋ (or, if you are not concerned about efficiency, or if you are using a DFA where it doesn't matter, you could use ⌈`\(([^()]|\([^()]*\))*\)`⌋ just as well), so the thought of having to worry about further levels of nesting is frightening.** Sometimes, you just have to use other, non-regex methods.

Watching Out for Unwanted Matches

It's very easy to forget what happens if the text is not formed just as you expect. Let's say you are writing a filter for converting a text file to HTML, and you want to replace a line of hyphens by `<HR>`, which represent a horizontal rule (a line across the page). If you used a `s/-*/<HR>/` search-and-replace command, it would replace the sequences you wanted, but only when they're at the beginning of the line. Surprised? In fact, `s/-*/<HR>/` will add `<HR>` to the beginning of *every* line, whether they begin with a sequence of hyphens or not!

Remember, anything that isn't required will always be considered successful. The first time ⌈`-*`⌋ is attempted at the start of the string, it matches any hyphens that are there. However, if there aren't any, it will still be happy to successfully match nothing. That's what star is all about.

Let's look at a similar example. I recently read a new book by a respected author in which he describes a regular expression to match a number, either an integer or floating-point. As his expression is constructed, such a number has an optional leading minus sign, any number of digits, an optional decimal point, and any number of digits that follow. His regex is ⌈`-?[0-9]*\.?[0-9]*`⌋.

Indeed, this matches such examples as `1`, `-272.37`, `129238843.`, `.191919`, and even something like `-. 0`. This is all good, and as expected.

---

* The use of ⌈`.*`⌋, should set off warning alarms to pay particular attention to decide whether dot is really what you want to apply star to. Sometimes that is exactly what you need, but ⌈`.*`⌋ is often used inappropriately.

** Here's a little Perl snippet that, given a $depth, creates a regex to match up to that many levels of parentheses beyond the first: `'\(' . '([^()]|\(' x $depth . '[^()]*' . '\))*' x $depth . '\)'` Analysis left to the reader.

However, how do you think it will match in `'this•test•has•no•number'`, `'nothing•here'`, or even an empty string? Look at the regex closely—*everything is* optional. *If* a number is there, and *if* it is at the beginning of the string, it will be matched, but *nothing is required.* This regex can match all three non-number examples, matching the nothingness at the beginning of the string each time. In fact, it even matches nothingness at the beginning of the string with an example like `'num•123'`, since that nothingness matches earlier than the number would.

So, it's important to say what you really mean. A floating-point number *must* have at least one digit in it, or it is not a number(!) To construct our regex, let's first assume there is at least one digit before the decimal point. If so, we need to use plus for those digits: ⌈`-?[0-9]+`⌋.

Writing the subexpression to match an optional decimal point (and subsequent digits) hinges on the realization that any numbers after the decimal point are contingent upon there being a decimal point in the first place. If we use something naive like ⌈`\.?[0-9]*`⌋, the ⌈`[0-9]*`⌋ gets a chance to match regardless of the decimal point's presence. The solution is, again, to say what we mean. A decimal point (and subsequent digits, if any) is optional: ⌈`(\.[0-9]*)?`⌋. Here, the question mark no longer quantifies (that is, governs) only the decimal point, but instead quantifies the combination of the decimal point plus any following digits. *Within* that combination, the decimal point is required; if it is not there, ⌈`[0-9]*`⌋ never even gets a chance to match.

Putting this all together, we have ⌈`-?[0-9]+(\.[0-9]*)?`⌋. This still doesn't allow something like `'.007'`, since our regex requires at least one digit before the decimal point. If we change the left side to allow zero digits, we will have to change the right side so it doesn't, since we can't allow all digits to be optional (the problem we were trying to correct in the first place).

The solution is to add an alternative which allows for the uncovered situation: ⌈`-?[0-9]+(\.[0-9]*)?|-?\.[0-9]+`⌋. This now also allows just a decimal point followed by (this time not optional) digits. Details, details. Did you notice that I allowed for the optional leading minus in the second alternative as well? That's easy to forget. Of course, you could instead bring the ⌈`-?`⌋ out of the alternation, as in ⌈`-?([0-9]+(\.[0-9]*)?|\.[0-9]+)`⌋.

Although this is an improvement on the original, you could still have problems, depending on how you use it. Often, a regex author has the context for the regex in mind, and so assumes something about the way it will be used, usually that it will be part of something larger where there are items on either side that disallow matches embedded in things that we really don't want to match. For example, our unadorned regex for floating-point numbers will be more than happy to match the string `'1997.04.12'`.

Yet, if we use this regex in a specific situation, such as in matching a data line where commas separate fields, wrapping our regex with ⌈`,…,`⌋, or perhaps even better ⌈`(^|,)…(,|$)`⌋, avoids embedding problems.

Matching Delimited Text

The IP address and ⌈`"[^"]*"`⌋ regexes we've examined are just two examples of a whole class of matching problem that often arises: the desire to match text delimited (or perhaps separated) by some other text. Other examples include:

- matching a C comment, which is delimited by '`/*`' and '`*/`'
- matching an HTML tag, which is text wrapped by `<…>`, such as `<CODE>`
- extracting items *between* HTML tags, such as the '`anchor text`' of the link '`<A•HREF="…">anchor text</A>`'
- matching a line in a *.mailrc* file. This file gives email aliases, where each line is in the form of

  `alias` *shorthand* *full-address*

  such as '`alias jeff jfriedl@ora.com`'. (Here, the delimiters are the whitespace between each item, as well as the ends of the line.)
- matching a quoted string, but allowing it to contain quotes if they are escaped, as in '`for your passport, you need a "2\"x3\" likeness" of yourself.`'

In general, the requirements for such tasks can be phrased along the lines of:

1. match the opening delimiter

2. match the main text
(which is really "match anything that is not the ending delimiter")

3. match the ending delimiter

As I mentioned earlier, the "match anything not the ending delimiter" can become complicated when the ending delimiter is more than one character, or in situations where it can appear within the main text.

Allowing escaped quotes in doublequoted strings

Let's look at the **`2\"x3\"`** example, where the ending delimiter is a quote, yet can appear within the main part if escaped.

The opening and closing delimiters are still the simple quotes, but matching the main text without matching the ending delimiter is the issue. Thinking clearly about which items the main text allows, we know that if a character is not a doublequote (in other words, if it's ⌈`[^"]`⌋), it is certainly okay. However, if it *is* a doublequote, it is okay if preceded by a backslash.

Unfortunately, there is no way to indicate "if preceded" in a regular expression. Were *lookbehind* possible, it would be helpful in exactly this kind of situation. Unfortunately, no one supports lookbehind yet. You *can* say "if followed" simply by appending what follows, or by using *lookahead* if supported (such as with Perl). There is no magic here; it is merely a matter of perspective. With the same view, ⌈`abc`⌋, becomes "`a` is okay if followed by `b` and then by `c`." This might sound like a silly distinction, but it is quite important. It means that we can't write "A doublequote is okay if preceded by a backslash." but we can say "A backslash followed by a doublequote is okay." That's written as ⌈`\\"`⌋. (Remember, a backslash is a metacharacter, so if we want our regex to match a literal backslash, we have to escape the backslash itself with another backslash.)

So, for the main-text part we have " okay if ⌈`[^"]`⌋, or if ⌈`\\"`⌋ yielding ⌈`[^"]|\\"`⌋. Since this is allowed any number of times until we get to the closing-delimiter quote, we apply with star. That requires parentheses, so putting it all together with the leading and trailing quotes gives us ⌈`"([^"]|\\")*"`⌋.

Sound logical? It does to me, but unfortunately, it doesn't work for two reasons. The first concerns only a Traditional NFA. When applying our regex to the string **`"2\"x3\" likeness"`**, we see that after the initial quote is matched, the first alternative is tried and matches the `2`. Due to the match, the alternation is left, but is immediately revisited because of the star. Again, the first alternative is tried first, and again it is able to match (the backslash). Unfortunately, this is a problem because that backslash should be recognized as the escape for the following doublequote, not a random "non-quote" of the ⌈`[^"]`⌋.

Continuing, the alternation is tried, but this time the first one fails since we're now at a doublequote. The second alternative, the escaped quote, fails as well since the escape is no longer available. The star therefore finishes, and the subsequent ending quote can match without problems. Therefore, we inadvertently match:

```
you need a "2\"x3\" likeness" of yourself.
```

This is not a problem when using a DFA or POSIX engine, as they always find the longest match from any particular starting point, period. (I might have mentioned this once or twice before.) These engines realize that if the escape is matched by ⌈`\\`⌋ and not ⌈`[^"]`⌋, the longer match (that we are expecting) is possible.

So, how do we solve the problem for a Traditional NFA? Well, if we switch the order of the two alternatives, ⌈`\\"`⌋ will be attempted before ⌈`[^"]`⌋ gets a chance to consume the escape, thus vaulting us past the escaped quote. So if we try ⌈`"(\\"|[^"])*"`⌋, it matches

```
you need a "2\"x3\" likeness" of yourself.
```

as we want. The first alternative usually fails (requiring the regex engine to retry with the second alternative), so extra backtracking is required when matching with this pattern, but at least it works.

Unfortunately, I said that there were two problems. The second problem, affecting all engine types, involves the situation where we almost have something that we want to match, but not quite. Consider:

```
"someone has \"forgotten\" the closing quote
```

Since there is no proper closing quote, we *do not* want this line to match at all. But as we've seen, an escaped quote *can* be matched. With POSIX and DFA engines and the previous examples, the escaped quote wasn't the final quote, so the match didn't end there. However, in this case, the escaped quote *is* the final quote, so the match does end there. Even with a Traditional NFA engineered to first pass over an escaped quote, the search for an overall match causes it to backtrack to find it.

This shows a particularly important moral: Always consider what will happen in the "odd" cases where you *don't* want your regex to match. For important situations, there is no substitute for really understanding what is happening, and for a comprehensive set of tests just in case.

Another moral: Make sure that there's no way that unwanted cases can sneak in through the back door. To get around our improper-match problem, we need to realize that both a doublequote and a backslash are "special" in this context, and that they must be handled separately from everything else. This means that the original dot, which became ⌜`[^"]`⌟, now changes again: ⌜`"(\\"|[^"\\])*"`⌟.

Since ⌜`[^"\\]`⌟ no longer causes the first problem, we can re-swap the alternatives to put the most likely case first: ⌜`"([^"\\]|\\")*"`⌟. This doesn't make any difference to a DFA (which doesn't backtrack) or a POSIX NFA (which must try all permutations, whichever order they happen to be in), but it increases efficiency for a Traditional NFA.

Allowing for other escaped items

One practical problem is that our regex can't match **`"hello, world\n"`** because the only backslash it allows is one before a quote. The solution is simply to change ⌈`\\"`⌋ to ⌈`\\.`⌋, which leaves us with ⌈`"([^"\\]|\\.)*`⌋

If the regex flavor's dot does not match newlines, you have a problem if you want this regex to allow escaped newlines. If the regex flavor supports ⌈`\n`⌋, you could use ⌈`(.|\n)`⌋ instead of the dot. A more efficient option (with tools that support it) is to use a character class that matches all bytes, such as ⌈`[\000-\377]`⌋). Note that ⌈`[.\n]`⌋ is a character class that matches the two characters period and newline (or, in some tools, a character class that matches two characters, period and `n`, and in still others, the three characters period, backslash, and `n`).

Knowing Your Data and Making Assumptions

This is an opportune time to highlight a general point about constructing and using regular expressions that I've briefly mentioned a few times. It is important to be aware of the assumptions made about the kind of data with which, and situations in which, a regular expression is intended to be used. Even something as simple as ⌈`a`⌋ makes an assumption that the target data is in the same character encoding (☞ 26) as the author intends. This is pretty much common sense, which is why I haven't been too picky about saying these things.

However, many assumptions that might seem obvious to one are not necessarily obvious to another. Our regex to match a doublequoted string, for example, assumes that there are no other doublequotes to consider. If you apply it to source code from almost any programming language, you might find, for instance, that it breaks because there can be doublequotes within comments.

There is nothing wrong with making assumptions about your data, or how you intend a regex to be used. The problems, if any, usually lie in overly optimistic assumptions and in misunderstandings between the author's intentions and how the regex is eventually used.

Additional Greedy Examples

There are certainly times when greediness works to your advantage. Let's look at a few simple examples. I'll use specific (and hopefully useful) applications to show general regular-expression construction techniques and thought processes.

With the thought that using is more interesting than reading, I'll sprinkle in a few examples coded in Perl, Tcl, and Python. If you're not interested in these particular languages, feel free to skip the code snippets. I'll use language-specific features, but still generally try to keep the examples simple and the lessons general.

Removing the leading path from a filename

The ability to manipulate filenames is often useful. An example is removing a leading path from a full pathname, such as turning `/usr/local/bin/gcc` into `gcc`.

Stating problems in a way that makes solutions amenable is half of the battle. In this case, we want to remove anything up to (and including) the final slash. If there is no slash, it is already fine and nothing needs to be done.

Here, we really do want to use ⌈`.*`⌋. With the regex ⌈`^.*/`⌋, the ⌈`.*`⌋ consumes the whole line, but then backs off (that is, backtracks) to the last slash to achieve the match. Since a slash is our substitute command delimiter, we have to either escape it within the regex, as with **`s/.*\///`** (the regular expression is marked), which could make one dizzy, or (if supported) use a different delimiter, say **`s!^.*/!!`**.

If you have a variable `$filename`, the following snippets ensure that there is no leading path:

Language	**Code Snippet**
Perl	`$filename =~ s!^.*/!!;`
Tcl	`regsub "^.*/" $filename "" filename`
Python	`filename = regsub.sub("^.*/", "", filename)`

By the way, if you work directly with DOS filenames, you would use a backslash instead of a slash. Since the backslash is a regular-expression metacharacter, you need to escape it (with another backslash). That would be something along the lines of `s/^.*\\//`. But with Tcl, Python, and other languages where regular expressions are just normal strings passed to regex-handling functions, backslash is not only a regex metacharacter, but it's often a string metacharacter as well. This means that you need `\\` just to get one backslash into the regex, and since you need two in the regex, you end up needing `\\\\` to match a literal backslash. Wow.

Remember this key point: Always consider what will happen if there is no match. In this case, if there is no slash in the string, no substitution is done and the string is left as is. Great, that's just what we want. . . a string such as '`/bin/sh`' becomes '`sh`', '`//../ernst`' becomes '`ernst`', and '`vi`' stays just as it is.

For efficiency's sake, it's important to remember how the regex engine goes about its work (if it is NFA-based, that is). Let's consider what happens if we omit the leading caret (something that's easy to do) and match against a string without a slash. As always, the regex engine starts the search at the beginning of the string. The ⌈`.*`⌋, races to the end of the string, but must back off to find a match for the slash. It eventually backs off everything that ⌈`.*`⌋ had gobbled up, yet there's still no match. So, the regex engine decides that there is no possible match *when starting from the beginning of the string*, but it's not done yet!

The transmission kicks in and retries the whole regex from the second character position. In fact, it needs (in theory) to go through the whole scan-and-backtrack routine for each possible starting position in the string. Filenames tend to be short, but the principle applies to many situations, and were the string long, that is potentially a lot of backtracking. Again, a DFA has no such problem.

In practice, a reasonable transmission realizes that any regex starting with ⌈`.*`⌋ that fails at the beginning of the string will never match when started from anywhere else, so it can shift gears and attempt the regex only the one time at the start of the string. Still, it's smarter to write that into our regex in the first place, as we originally did.

Accessing the filename from a path

A related option is to bypass the path and simply match the trailing filename part without the path, putting that text into another variable. The final filename is everything at the end that's not a slash: ⌈`[^/]*$`⌋. This time, the anchor is not just an optimization; we really do need dollar at the end. We can now do something like:

```
    $WholePath =~ m!([^/]*)$!;     # check variable $WholePath
with regex.
    $FileName = $1;                # note text matched
```

You'll notice that I don't check to see whether the regex actually matches or not, because I *know* it will match every time. The only *requirement* of that expression is that the string have an end to match dollar, and even an empty string has an end.

Thus, when I use `$1` to reference the text matched within the parenthetical subexpression, I'm assured it will have some (although possibly empty) value.*

However, since Tcl, Python, and Perl all use NFAs (Traditional NFAs, to be specific), ⌈`[^\/]*$`⌋ is very inefficient. Carefully run through how the NFA engine attempts the match and you see that it can involve a lot of backtracking. Even the short sample '`/usr/local/bin/perl`' backtracks over 40 times before finally matching.

Consider the attempt that starts at `local/`. Once ⌈`[^/]*`⌋ matches through to the second `l` and fails on the slash, the ⌈`$`⌋ is tried (and fails) for each `l`, `a`, `c`, `o`, `l` saved state. If that's not enough, most of it is repeated with the attempt that starts at `local/`, and then again `local/`, and so on.

It shouldn't concern us too much with this particular example, as filenames tend to be short. (And 40 backtracks is nothing   40 million is when they really matter!) Again, since it's important to be aware of the issues, the general lessons here can be applied to your specific needs.

This is a good time to point out that even in a book about regular expressions, regular expressions aren't always The Answer. Tcl, for example, provides special commands to pick apart pathnames (part of the `file` command set). In Perl,

```
$name = substr($WholePath, rindex($WholePath, "/")+1);
```

is much more efficient. However, for the sake of discussion, I'll forge ahead.

---

* If you're familiar with Perl, you might wonder why I used parentheses and `$1` instead of just using `$&` (a variable representing the overall text matched). The reason is that there's a potentially large performance hit for a program that uses `$&` see "Unsociable $& and Friends" (☞ 273).

Both leading path and filename

The next logical step is to pick apart a full path into both its leading path and filename component. There are many ways to do this, depending on what we want. Initially, you might want to use ⌈`^(.*)/(.*)$`⌋ to fill `$1` and `$2` with the requisite parts. It looks like a nicely balanced regular expression, but knowing how greediness works, we are guaranteed that the first ⌈`.*`⌋ will never leave anything with a slash for `$2`. The only reason the first ⌈`.*`⌋ leaves anything at all is due to the backtracking done in trying to match the slash that follows. This leaves only that "backtracked" part for the later ⌈`.*`⌋. Thus, `$1` will be the full leading path and `$2` the trailing filename.

One thing to note: we are relying on the initial ⌈`(.*)/`⌋ to ensure that the second ⌈`(.*)`⌋ does not capture any slash. We understand greediness, so this is okay. Still I like to be specific when I can, so I'd rather use ⌈`[^/]*`⌋ for the filename part. That gives us ⌈`^(.*)/([^/]*)$`⌋. Since it shows exactly what we want, it acts as documentation as well.

One big problem is that this regex requires at least one slash in the string, so if we try it on something like `file.txt`, there's no match, and no information. This can be a feature if we deal with it properly:

```
if ( $WholePath =~ m!^(.*)/(.*)$! ) {
    $LeadingPath = $1;
    $FileName = $2;
} else {
    $LeadingPath = ".";   #  so "file.txt" looks like ". /file.txt"
    $FileName = $WholePath;
}
```

Another method for getting at both components is to use either method for accessing the path or file, and then use the side effects of the match to construct the other. Consider the following Tcl snippet which finds the location of the last slash, then plucks the substrings from either side:

```
    if [regexp -indices .*/ $WholePath Match] {
       #  We have a match. Use the index to the end of the match to find the slash.
       set LeadingPath [string range $WholePath 0 [expr [lindex
$Match 1] -1]]
       set FileName [string range $WholePath [expr [lindex
$Match 1] +1] end]
    } {
       #  No match - whole name is the filename.
       set LeadingPath .
       set FileName $WholePath
    }
```

Here, we use Tcl's `regexp -indices` feature to get the index into `WholePath` of the match: if the string is `/tmp/file.txt`, the variable `Match` gets '`0 4`' to reflect that the match spanned from character `0` to character `4`. We know the second index points to the slash, so we use **`[expr [lindex $Match 1] - 1]`** to point just before it, and a `+1` version to point just after it. We then use string range to pluck the two substrings.

Again, with this particular example, using a regex to find the last slash is sort of silly some kind of an rindex function would be faster (in Tcl, it would be **`string last / $WholePath`**). Still, the idea of plucking out parts of the string this way is intriguing, even if this simple example can be done differently.

Summary

If you understood everything in this chapter the first time reading it, you probably didn't need to read it in the first place. It's heady stuff, to say the least. It took me quite a while to understand it, and then longer still to *understand* it. I hope this one concise presentation makes it easier for you. I've tried to keep the explanation simple without falling into the trap of oversimplification (an unfortunately all-tocommon occurrence which hinders a real understanding).

This chapter is divided roughly into two parts—a description of match mechanics, and some of their practical effects.

Match Mechanics Summary

There are two underlying technologies commonly used to implement a regex match engine, "regex-directed NFA" (☞ 99) and "text-directed DFA" (☞ 100) [abbreviations spelled out ☞ 101].

Combine the two technologies with the POSIX standard (☞ 116), and for practical purposes there are three types of engines:

•	Traditional NFA	(gas-guzzling, power-on-demand) engine
•	POSIX NFA	(gas-guzzling, standard-compliant) engine
•	DFA (POSIX or not)	(electric, steady-as-she-goes) engine

To get the most out of a utility, you need to understand which type of engine it uses, and craft your regular expressions appropriately. The most common type is the Traditional NFA, followed by the DFA. Table 4-1 (☞ 90) lists a few common tools and their engine type. In Chapter 5's "Testing the Engine Type" (☞ 160), I show how you can test for the engine type yourself.

One overriding rule regardless of engine type: matches starting sooner take precedence over matches starting later. This is due to how the engine's "transmission" tests the regex at each point in the string (☞ 92).

For the match attempt starting at any given spot

DFA Text-Directed Engines

Find the longest possible match, period. That's it. End of discussion (☞ 115). Consistent, Very Fast, and Boring to talk about (☞ 118).

NFA Regex-Directed Engines

Must "work through" a match. The soul of NFA matching is *backtracking* (☞ 102, 106). The metacharacters control the match: the quantifiers (star and friends) are *greedy* (☞ 94). Alternation is not usually greedy (☞ 112), but is with a POSIX NFA.

POSIX NFA Must find the longest match, period. But it's not boring, as you must worry about efficiency (the subject of the next chapter).

Traditional NFA Can be the most expressive regex engine, since you can use the regex-directed nature of the engine to craft exactly the match you want.

"DFA and NFA in Comparison" on page 118 summarizes the differences among the engine types.

We never did find out the regex counterpart to *Riiich Coriiiiinthian Leaaaather*.

Some Practical Effects of Match Mechanics

Matching delimited text, such as doublequoted strings or C comments is a common task (☞ 129). The general approach is to match the opening delimiter, then anything not the closing delimiter, and finally the closing delimiter. The difficulty usually lies in ensuring that the closing delimiter doesn't sneak into the middle phase. Be sure to understand how persistent greediness can be (☞ 110).

Greediness is your friend if used skillfully, but it can lead to pitfalls if you're not careful. It's a good idea to be as specific as you can (☞ 122), and to pay careful attention to how unwanted matches might sneak in (☞ 127).

Constructing a regex for a specific task often requires striking a balance among matching what you want, not matching what you don't want, and (for an NFA) being efficient (☞ 121). For an NFA, efficiency is so crucial that I devote the next chapter to crafting an efficient NFA regular expression.

5

Crafting a Regular Expression

In this chapter

- ***A Sobering Example***
- ***A Global View of Backtracking***
- ***Internal Optimizations***
- ***Testing the Engine Type***
- ***Unrolling the Loop***
- ***Unrolling C Comments***
- ***The Freeflowing Regex***
- ***Think!***

With the regex-directed nature of an NFA engine, as is found in Perl, Tcl, Expect, Python, and some versions of *grep, awk, egrep,* and *sed* (just to name a few), subtle changes in an expression can have major effects on what or how it matches. Issues that simply don't matter with a DFA engine become paramount. The fine control an NFA engine affords allows you to really *craft* an expression, although it can sometimes be a source of confusion to the unaware. This chapter will help you learn this art.

At stake are both correctness and efficiency. This means matching just what you want and no more, and doing it quickly. The previous chapter examined correctness; here we'll look at efficiency issues of an NFA engine and how to make them work to our advantage. (DFA-related issues will be mentioned when appropriate, but this chapter is primarily concerned with NFA-based engines and their efficiency.) In a nutshell, the keys are understanding the full implications of backtracking and learning techniques to avoid it where possible. We'll look at some techniques for writing efficient expressions that will not only help with efficiency, but armed with the detailed understanding of the processing mechanics, you will also be able to write more complex expressions with confidence.

Toward arming you well, this chapter first presents a rather detailed example illustrating just how important these issues can be, then prepares you for some of the more advanced techniques presented later by reviewing the basic backtracking described in the previous chapter with a strong emphasis on efficiency and backtracking's global ramifications. This is followed by a look at some of the common internal optimizations that can have a fairly substantial impact on efficiency, and on how expressions are best written for implementations that employ them. Finally, I bring it all together with some killer techniques to construct lightning-fast NFA regexes.

Tests and Backtracks

As in most chapters, these examples merely illustrate common situations that are met when using regular expressions. When examining a particular example's efficiency, I'll often report the number of individual tests that the regex engine does during the course of a match. For example, in matching ⌈`marty`⌋ against `smarty`, there are six individual tests—the initial attempt of ⌈`m`⌋ against `s` (which fails), then the matching of ⌈`m`⌋ against `m`, ⌈`a`⌋ against `a`, and so on (all of which are successful). *I* also often report the number of backtracks (one in this example—the implicit backtrack to retry the regex at the second character position).

I use these exact numbers not because the precision is important, but rather to be more concrete than words such as "lots," "few," "many," "better," "not too much," and so forth. I don't want to imply that using regular expressions with an NFA is an exercise in counting tests or backtracks; I just want to acquaint you with the relative qualities of the examples.

Another important thing to realize is that these "precise" numbers probably differ from tool to tool. It's the basic relative performance of the examples that I hope will stay with you. One important variable, however, is the optimizations a tool might employ. A smart enough implementation could completely bypass the application of a particular regex if it can decide beforehand that it could not possibly match the string in question (in cases, for instance, when the string lacks a particular character that the engine knows beforehand will be required for any possible match). I discuss these important optimizations in this chapter, but the overall lessons are generally more important than the specific special cases.

Traditional NFA vs. POSIX NFA

It is important to keep in mind the target tool's engine type, Traditional NFA or POSIX NFA, when analyzing efficiency. As we'll see in the next section, some concerns matter to only one or the other. Sometimes a change that has no effect on one has a great effect on the other. Again, understanding the basics allows you to judge each situation as it arises.

A Sobering Example

Let's start by looking at an example that really shows how important of a concern backtracking and efficiency can be. Toward the end of Chapter 4, we came up with 「`"(\\.|[^"\\])*"`」 to match a quoted string, with internal quotes allowed if escaped (☞ 129). This regex works, but if it's used with an NFA engine, the alternation applied at each character is very inefficient. With every "normal" (non-escape, non-quote) character in the string, the engine has to test 「`\\.`」, fail, and backtrack to finally match with 「`[^"\\]`」 If used where efficiency matters, we would certainly like to be able to speed this regex up a bit.

A Simple Change—Placing Your Best Foot Forward

Since the average doublequoted string has more normal characters than escaped ones, one simple change is to swap the order of the alternatives, putting 「`[^"\\]`」 first and 「`\\.`」 second. By placing 「`[^"\\]`」 first, alternation backtracking need be done only when there actually is an escaped item in the string (and once for when the star fails, of course, since all alternatives must fail for the alternation as a whole to fail). Figure 5-1 illustrates this difference visually. The reduction of arrows in the bottom half represents the increased number of times when the first alternative matches. That means less backtracking.

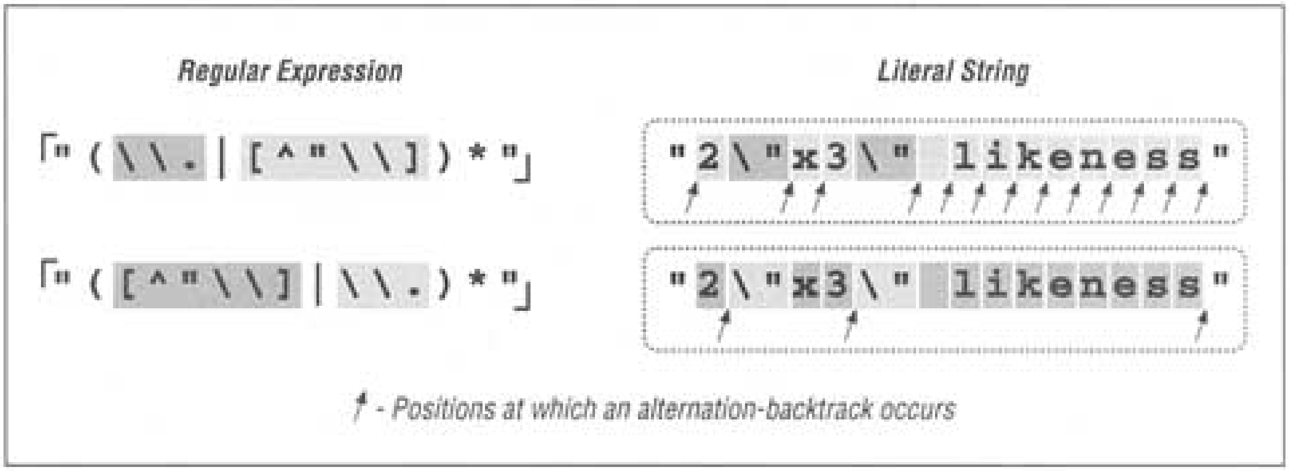


Figure 5-1:
Effects of alternate order (Traditional NFA)

You should ask certain key questions whenever evaluating a change for efficiency's sake, and I'll put two to you now:

- Will this change benefit a Traditional NFA, POSIX NFA, or both?

- Will this change be of most benefit when the text matches, when the match fails, or at all times?

❖ Please consider these issues carefully before flipping the page to check your answers, and make sure that you have a good grasp of the answers (and reasons) before continuing on to the next section.

More Advanced—Localizing the Greediness

Looking at Figure 5-1, it's very clear that for either case star must iterate (or cycle, if you like) for each normal character, entering and leaving the alternation (and the parentheses) over and over. These actions involve overhead, which means extra work—extra work that we'd like to eliminate if possible.

Once while working on a similar expression, I realized that I could make an optimization by taking into account that ⌈`[^\\]`⌋ is the normal case. Using ⌈`[^"\\]+`⌋ instead allows one iteration of `(…)*` to read as many normal (non-quote, non-

Effects of a simple change

❖ Answers to the questions on page 141.

Effect for which type of engine? The change has virtually no effect whatsoever for a POSIX NFA engine. Since it must eventually try every permutation of the regex anyway, the order in which the alternatives are tried is insignificant. For a Traditional NFA, however, ordering the alternatives in such a way that quickly leads to a match is a benefit because the engine can stop once the first match is found.

Effect during which kind of result? The change results in a faster match only when there *is* a match. An NFA can fail only after trying all possible permutations of the match (and again, the POSIX NFA tries them all anyway). So if indeed it ends up failing, every permutation must have been attempted, so the order does not matter.

The following table shows the number of tests ("tests") and backtracks ("b.t.") for several cases (smaller numbers are better):

	Traditional NFA				**POSIX NFA**	
	⌈`"(\\.|[^"\\])*"`⌋		⌈`"([^"\\]|\\.)*"`⌋		*either*	
Sample String	**tests**	**b.t.**	**tests**	**b.t.**	**tests**	**b.t.**
`"2\"x3\" likeness"`	32	14	22	4	48	30
`"makudonarudo"`	28	14	16	2	40	26
`"very`…99 *more chars* …`long"`	218	109	111	2	325	216
`"No \"match\" here`	124	86	124	86	124	86

As you can see, the POSIX NFA results are the same with both expressions, while the Traditional NFA's performance increases (backtracks decrease) with the new expression. Indeed, in a non-match situation (the last example in the table), since both types of engine must evaluate all possible permutations, all results are the same.

escape) characters as there are in a row. For strings without any escapes, this would be the entire string. This allows a match with almost no backtracking, and also reduces the star iteration to a bare minimum. I was very pleased with myself for making this discovery.

We'll look at this example in more depth later in this chapter, but a quick look at some statistics clearly shows the benefit. Figure 5-2 looks at this example for a Traditional NFA. Applying this new change to the original ⌈`"(\\.|[^"\\])*"`⌋ (the upper pair of Figure 5-2), alternation-related backtracks and star iterations are both reduced. The lower pair in Figure 5-2 illustrates that performance is enhanced even more when this change is combined with our previous reordering.

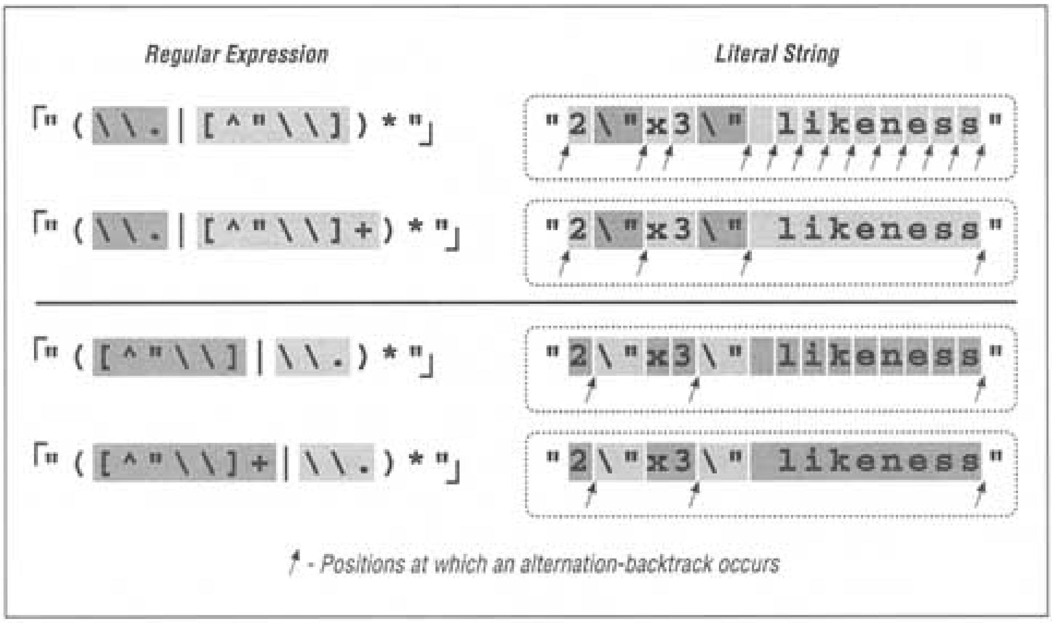

Regular Expression
Literal String
⌈"(\\.|[^"\\])*"⌋
"2\"x3\" likeness"
⌈"(\\.|[^"\\]+)*"⌋
"2\"x3\" likeness"
⌈"([^"\\]|\\.)*"⌋
"2\"x3\" likeness"
⌈"([^"\\]+|\\.)*"⌋
"2\"x3\" likeness"
- Positions at which an alternation-backtrack occurs

Figure 5-2:
Effects of an added plus (Traditional NFA)

The big gain with the addition of plus is the resulting reduction in the number of alternation backtracks, and, in turn, the number of iterations by the star. The star quantifies a parenthesized subexpression, and each iteration entails a fair amount of overhead as the parentheses are entered and exited, as the engine needs to keep tabs on what text is matched by the enclosed subexpression. (This is discussed in depth later in this chapter.)

Table 5-1 is similar to the one in the answer block, but it addresses a smaller set of situations and adds information about the number of iterations required by star. In each case, the number of individual tests and backtracks increases ever so slightly, but the number of cycles is drastically reduced. This is big savings.

Table 5-1: Match Efficiency for a Traditional NFA

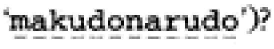
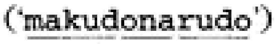
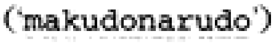
	⌈`"([^"\\]|\\.)*"`⌋			⌈`"([^"\\]±|\\.)*"`⌋		
Sample String	tests	b.t.	*-cycles	tests	b.t.	*-cycles
`"makudonarudo"`	16	2	13	17	3	2
`"2\"x3\" likeness"`	22	4	15	25	7	6
`"very`… *99 more chars* …`long"`	111	2	108	112	3	2

Reality Check

Yes, I was quite pleased with myself for this discovery. However, as wonderful as this "enhancement" might seem, it was really a monster in disguise. You'll notice that when extolling its virtues, I did not give statistics for a POSIX NFA engine. If I had, you might have been surprised to find the `"very•...•long"` example required over *three hundred thousand million billion trillion* backtracks (for the record, the actual count would be 324,518,553,658,426,726,783,156,020,576,256, or about 325 nonillion—if I had a nickel for each one, I'd be almost as rich as Bill Gates). Putting it mildly, that is a LOT of work. On my system, this would take well over *50 quintillion* years, take or leave a few hundred trillion millennia.*

Quite surprising indeed! So, why does this happen? Briefly, it's because something in the regex is subject to both an immediate plus and an enclosing star, with nothing to differentiate which is in control of any particular target character. The resulting nondeterminism is the killer. Let me explain a bit more.

Before adding the plus, ⌜`[^"\\]`⌟ was subject to only the star, and the number of possible ways for the effective ⌜`[^\\]*`⌟ to divvy up the line was limited. It could match one character, two characters, etc., but the number of possibilities was directly proportional to the length of the target string.

With the new regex's effective ⌜`([^"\\]+)*`⌟, the number of ways that the plus and star might divvy up the string explodes exponentially. If the target string is `makudonarudo`, should it be considered 12 iterations of the star, where each internal ⌜`[^"\\]+`⌟ matches just one character (as might be shown by  'makudonarudo')? Or perhaps one iteration of the star, where the internal ⌜`[^"\\]+`⌟ matches everything ('makudonarudo')? Or, perhaps 3 iterations of the star, where the internal ⌜`[^"]\\]+`⌟ match 5, 3, and 4 characters respectively  ('makudonarudo'). Or perhaps 2, 7, and 3 characters respectively  ('makudonarudo'). Or, perhaps. . .

Well, you get the idea—there are a lot of possibilities (4,096 in this 12-character example). For each extra character in the string, the number of possible combinations doubles, and the POSIX NFA must try them all before returning its answer. That means backtracking, and lots[**] of it! Twelve character's 4,096 combinations doesn't take long, but 20 character's million-plus combinations take more than a few seconds. By 30 characters, the trillion-plus combinations take hours, and by 40, it's well over a year. Obviously, this is not acceptable.

---

[*] I use an IBM ThinkPad 755cx with a 75MHz Pentium, running Linux. Note that the reported time is estimated based upon other benchmarks; I did not actually run the test that long.

[**] For those into such things, the number of backtracks done on a string of length *n* is 2^{n+1}. The number of individual tests is $2^{n+1}+ 2^{n}$.

"Ah," you might think, "but a POSIX NFA is not yet all that common. I know my tool uses a Traditional NFA, so I'm okay." The major difference between a POSIX and Traditional NFA is that the latter stops at the first full match. If there is no full match to be had, even a Traditional NFA must test every possible combination before it finds that out. Even in the short `"No•\"match\"•here` example from the previous answer block, 8,192 combinations must be tested before the failure can be reported.

Yes, I had been quite pleased with myself for the trick I discovered, but also thought I found a bug in the tool when it would sometimes seem to "lock up." It turns out that it was just crunching away on one of these neverending matches. Now that I understand it, this kind of expression is part of my regular-expression benchmark suite, used to indicate the type of engine a tool implements:

- If one of these regexes is fast even with a non-match, it's a DFA.
- If it's fast only when there's a match, it's a Traditional NFA.
- If it's slow all the time, it's a POSIX NFA.

I talk about this in more detail in "Testing the Engine Type" on page 160.

Certainly, not every little change has the disastrous effects we've seen with this example, but unless you know the work going on behind an expression, you will simply never know until you run into the problem. Toward that end, this chapter looks at the efficiency concerns and ramifications of a variety of examples. As with most things, a firm grasp of the underlying basic concepts is essential to an understanding of more advanced ideas, so before looking at ways to solve this neverending match problem, I'd like to review backtracking in explicit detail.

A Global View of Backtracking

On a local level, backtracking is returning to attempt an untried option. On a global level, backtracking is not packaged so neatly. In this section, we'll take an explicit look at the details of backtracking both during a match and during a non-match, and we'll try to make some sense out of the patterns we see emerge. If the reality check of the last section didn't surprise you, and you feel comfortable with these details, feel free to skip ahead to "Unrolling the Loop" where we'll look at some advanced efficiency techniques.

Let's start by looking closely at some examples from the previous chapter. First, if we apply ⌜`".*"`⌟ to

```
The name "McDonald's" is said "makudonarudo" in Japanese
```

we can visualize the matching action as shown in Figure 5-3 on the next page.

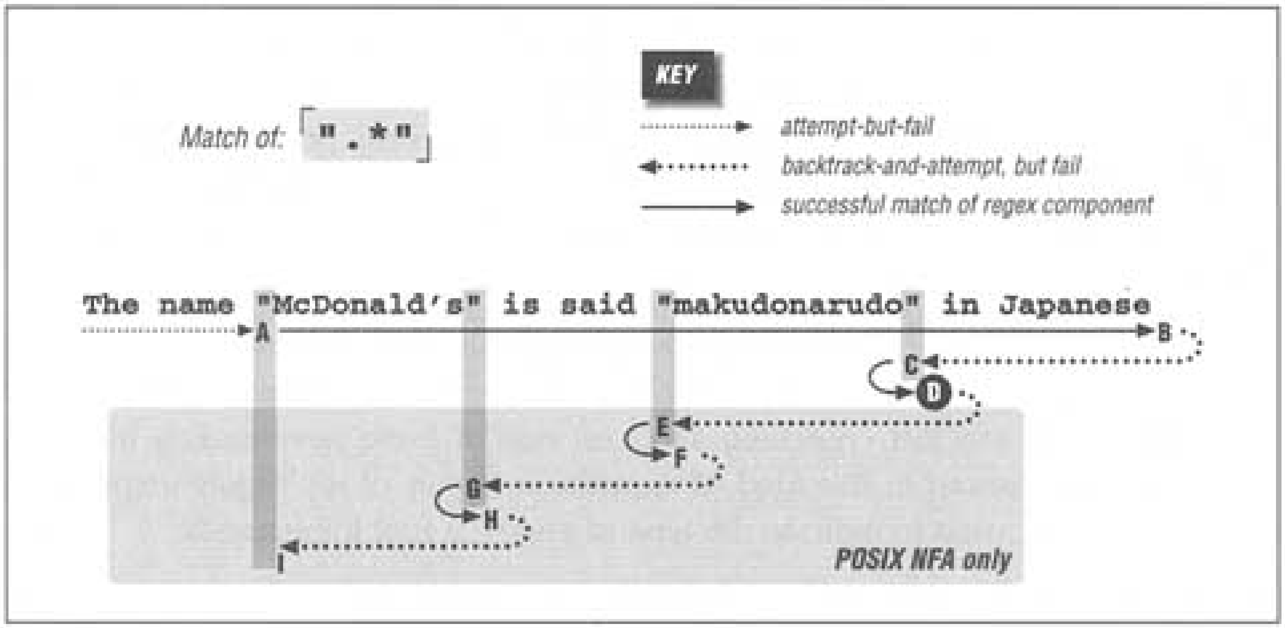

Match of: 「".*"」
KEY
attempt-but-fail
backtrack-and-attempt, but fail
successful match of regex component
The name "McDonald's" is said "makudonarudo" in Japanese
A
B
C
D
E
F
G
H
I
POSIX NFA only

Figure 5-3:
Successful match of 「".*"」

The regex is attempted starting at each string position in turn, but because the initial quote fails immediately, nothing interesting happens until the attempt starting at the location marked **A**. At this point, the rest of the expression is attempted, but the transmission (☞ 92) knows that if the attempt turns out to be a dead end, the full regex can still be tried at the next position.

The 「.*」 then matches to the end of the string, where the dot is unable to match the nothingness at the end of the string and so the star finally stops. None of the 46 characters matched by 「.*」 is required, so while matching them, the engine accumulated 46 more situations to where it can backtrack if it turns out that it matched too much. Now that 「.*」 has stopped, the engine backtracks to the last of those saved states, the " try ".*" at anese" **st**ate.

This means that we try to match the closing quote at the end of the string. Well, a quote can match nothingness no better than dot, so this fails too. The engine backtracks again, this time trying to match the closing quote at Japanese, which also fails.

The remembered states accumulated while matching from **A** to **B** are tried in reverse (latest first) order as we move from **B** to **C**. After trying only about a dozen of them, the state that represents "try `".*"`" at "`...arudo" in Japa...`" is reached, point **C**. This *can* match, bringing us to **D** and an overall match:

```
The name "McDonald's" is said "makudonarudo" in Japanese
```

If this is a Traditional NFA, the remaining unused states are simply discarded and the successful match is reported.

More Work for a POSIX NFA

For POSIX NFA, the match noted above is remembered as "the longest match we've seen so far," but all remaining states must be explored to see whether they could come up with a longer match. We know this won't happen in this case, but the regex engine must find that out for itself. The states are tried and immediately discarded except for the remaining two situations where there is a quote in the string available to match the final quote. Thus, the sequences **`D-E-F`** and **`F-G-H`** are similar to **`B-C-D`**, except the matches at **`F`** and **`H`** are discarded as being shorter than a previously found match.

By **`I`**, the only remaining backtrack is the "bump along and retry" one. However, since the attempt starting at **`A`** *was* able to find a match (three in fact), the POSIX NFA engine is finally done.

Work Required During a Non-Match

We still need to look at what happens when there is no match. Let's look at ⌈`".*"!`⌋, which we know won't match our example text. It comes close on a number of occasions throughout the match attempt, resulting in, as we'll see, much more work.

Figure 5-4 on the next page illustrates this work. The **`A-I`** sequence looks similar to that in Figure 5-3. One difference is that this time it does not match at point **`D`** (because the ending exclamation point can't match). Another difference is that the entire sequence in Figure 5-4 applies to both Traditional and POSIX NFA engines: finding no match, the Traditional NFA must try as many possibilities as the POSIX NFA all of them.

Since there is no match from the overall attempt starting at **`A`** and ending at **`I`**, the transmission bumps along to retry the match. Attempts eventually starting at points **`J`**, **`Q`**, and **`V`** look promising, but fail similarly to the attempt at **`A`**. Finally at **`Y`**, there are no more positions for the transmission to try from, so the overall attempt fails. As Figure 5-4 shows, it took a fair amount of work to find this out.

Being More Specific

As a comparison, let's replace the dot with ⌈`[^"]`⌋. As discussed in the previous chapter, this gives less surprising results because it is more specific, and is more efficient to boot. With ⌈`"[^"]*"!`⌋, the ⌈`[^"]*`⌋ can't get past the closing quote, eliminating much matching and subsequent backtracking.

Figure 5-5 on page 149 shows the failing attempt (compare to Figure 5-4). As you can see, much less backtracking is needed. If the different results suit your needs, the reduced backtracking is a welcome side effect.

Alternation Can Be Expensive

Alternation can be a leading causes of backtracking. As a simple example, let's use our `makudonarudo` test string to compare how 「`u|v|w|x|y|z`」 and 「`[uvwxyz]`」 go about matching. A character class is a simple test, so 「`[uvwxyz]`」 suffers only the bump-along backtracks (34 of them) until we match at:

```
      The name "McDonald's" is said "makudonarudo"  in Japanese
```

With 「`u|v|w|x|y|z`」, however, there need to be six backtracks at each starting position, eventually totaling 204 before we achieve the same match.

Obviously, not every alternation is replaceable, and even if so, it's not necessarily as easily as with this example. In some situations, however, certain techniques that we'll look at later can greatly reduce the amount of alternation backtracking required for a match.

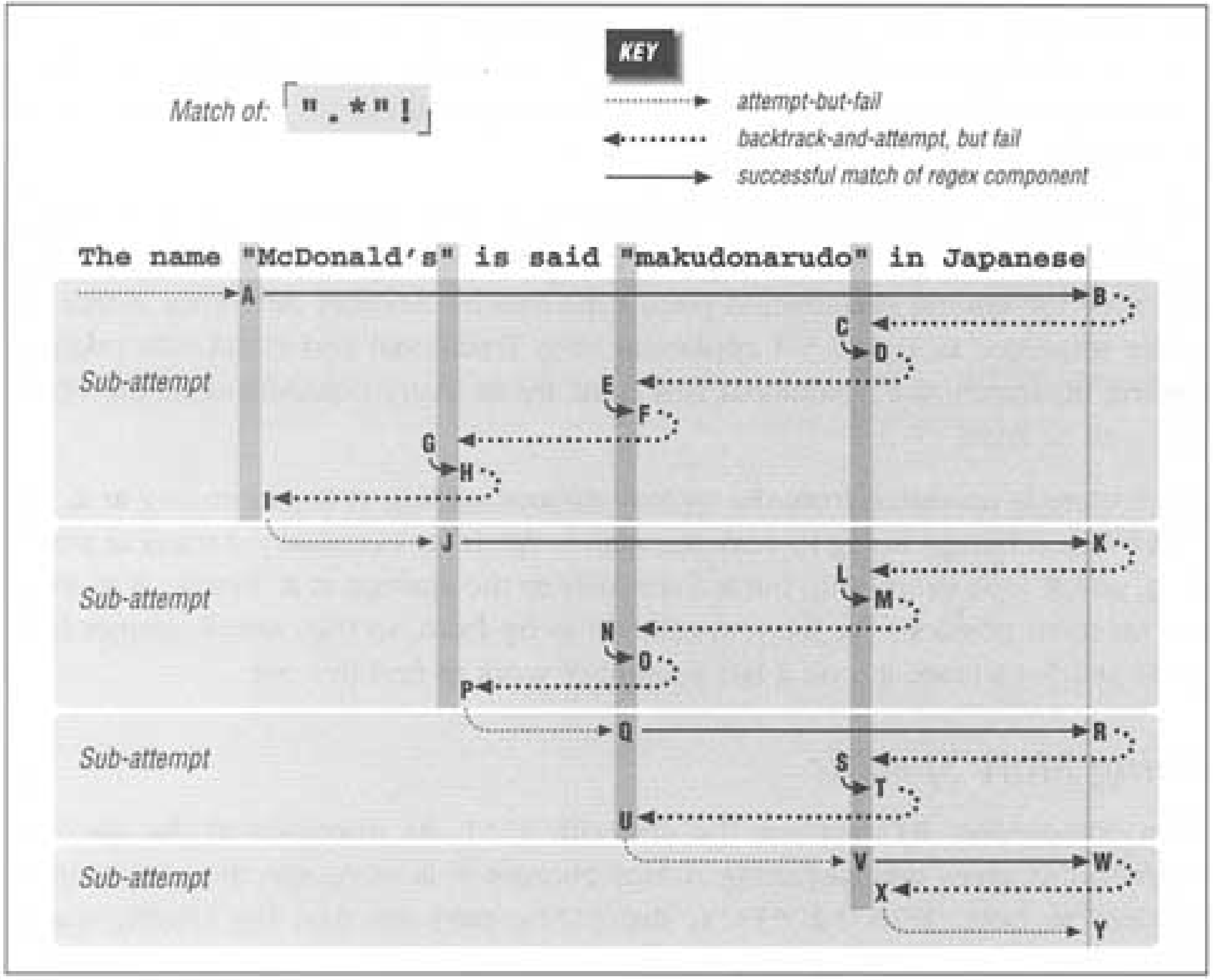

Match of: ⌜".*"!⌟
KEY
attempt-but-fail
backtrack-and-attempt, but fail
successful match of regex component
The name "McDonald's" is said "makudonarudo" in Japanese
Sub-attempt
Sub-attempt
Sub-attempt
Sub-attempt
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P
Q
R
S
T
U
V
W
X
Y

Figure 5-4:
Failing attempt to match ⌜".*"!⌟

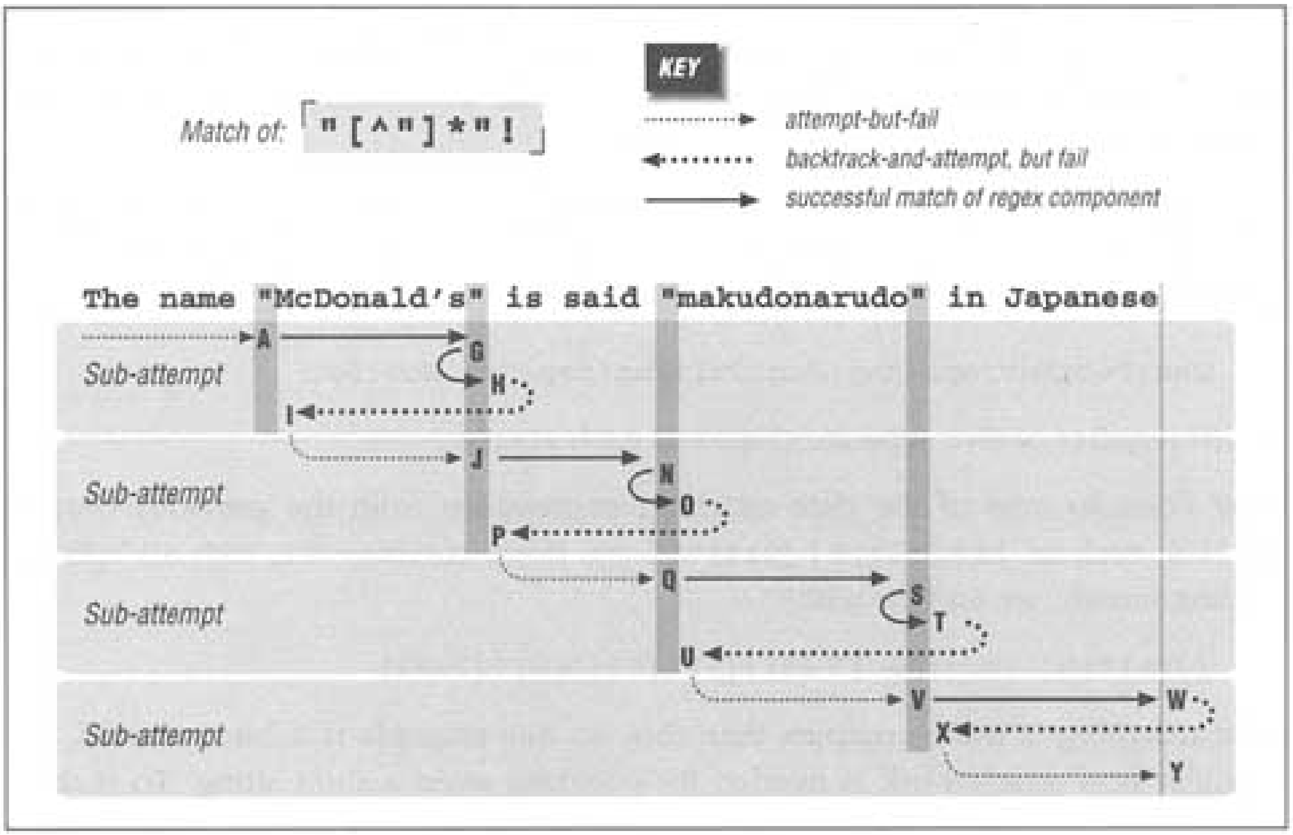

Match of: ⌈"[^"]*"!⌋
KEY
attempt-but-fail
backtrack-and-attempt, but fail
successful match of regex component
The name "McDonald's" is said "makudonarudo" in Japanese
Sub-attempt
A
G
H
I
Sub-attempt
J
N
O
P
Sub-attempt
Q
S
T
U
Sub-attempt
V
W
X
Y

Figure 5-5:
Failing attempt to match ⌈"["]*"!⌋

A Strong Lead

From an efficiency standpoint, a good thing about the earlier examples in this chapter is that the expressions begin with a single, simple element (the quote). If it doesn't match, the whole attempt (at the current starting position) is aborted immediately. This, in turn, allows the transmission to quickly bump along to retry at the next position.

As we see with alternation (among other situations), not all expressions are this efficient. For example, to match a simple singlequoted or doublequoted string, a first draft might be 「`'[^']*'|"[^"]*"`」. With this, each match attempt starts not with the simple quote, but with an effective 「`'|"`」, because both alternatives must be checked. This means backtracking. It would be nice if the regex engine realized that any match must begin with one of the quotes, and not even bother attempting the whole expression except when starting on a quote. In fact, some engines do this optimization automatically if they can figure out what the first character can be. With a special feature of Perl called *lookahead*,[*] you can manually have 「`['"]`」 checked "off to the side" to immediately fail if the rest of the expression has no possible chance to match. By the way, if the regex were ⌈`'.*'|".*"`⌋ instead, we could use ⌈`(['"]).*\1`⌋, but you see that this technique can't be used with our example, since `\1` can't be used within a negated (or any kind of) character class.

---

[*] Perl's lookahead construct is 「`(?=...)`」, so to do a "precheck" for 「`['"]`」, we would insert 「`(?=['"])`」 at the start of the regex. Testing this on a variety of data, I found that the added check cut the time by 20 to 30 percent. With the month example that follows shortly, adding 「`(?=[ADFJMNOS])`」 cuts the time by a good 60 percent.

This might seem to be a lot of fuss over nothing, since simply checking two alternatives at each position in the string doesn't seem to be much work. However, consider something such as

⌈`Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|Nov|Dec`⌋

which requires twelve separate checks at each attempt.

Now consider one of the date-matching expressions from the previous chapter (☞ 114), such as ⌈`31|[123]0|[012]?[1-9]`⌋. If we combine this with an optional leading month, we end up with:

⌈`(Jan|Feb|…|NovDec)? (31|[123]0|[012]?[1-9])`⌋

Date matching is more complex than this, so this example is a bit contrived, but consider how much work is needed for checking even a short string. To make a check from any particular starting position, each of the month alternatives is checked in turn. Then, the date alternatives must be tried. Only after they have all been tried and all fail can the engine move on to the next starting position. On top of all the backtracking, the parentheses incur additional overhead.

The Impact of Parentheses

Although not directly related to backtracking, one important factor of efficiency is the capturing overhead related to the number of times that a set of parentheses is entered and exited. For example, with ⌈`"(.*)"`⌋, once the initial quote has matched, the set of parentheses wrapping ⌈`.*`⌋ is entered. In doing this, the regex engine does some housekeeping in preparation for saving the text matched until the parentheses are exited. Internally, it is more complicated than it appears at first glance, since the "current status" of each set of parentheses needs to be part of the backtracking state maintained by the engine.

Let's look at four different expressions that match the same string:

	Regex	Item Captured Within Parentheses
1.	`".*"`	
2.	`(".*)`	Entire string (with quotes)
3.	`"(.*)"`	Body of string (without quotes)
4.	`"(.)*"`	Last character of string (before final quote)

The differences lie in what is matched within the parentheses, and the efficiency with which a match is done. With these examples, the parentheses add nothing to the logic of the expression itself, so they are used apparently only to capture text. In such cases, the needs of the particular problem should dictate their use, but for the purposes of discussion, I'd like to explore the relative performance issues.

A detailed look at the performance effects of parentheses and backtracking

Let's first consider the overhead required for the basic "entering the parentheses" setup for the average case when an attempt is started at something other than a quote (that is, at each position where the match can't even get off the ground). With *#3*, and *#4*, the parentheses are not reached until the first quote matches, so there is no parentheses-related overhead at all. With *#2*, though, the parentheses are entered at each attempt, and only then does the requirement for a quote to match cause the attempt to fail. This means wasted work, even when the match can't get off the ground. However, a *first-character-discrimination* optimization, discussed in the next section, can be used to bypass this wasted overhead. If the transmission realizes that a match must begin with a quote to have any chance of being successful, it can quickly bypass match attempts that don't begin at a quote.

Even if there is no optimization, *#2's* one extra enter-the-parentheses overhead per attempt is not such a big deal because it doesn't involve inter-parentheses backtracking, nor any exiting. A much larger issue is the overhead in example *#4*, where the parentheses are entered and exited at *each* character of the string.

Furthermore, because ⌈`(.)*`⌋ first matches all the way to the end of the line (or string, depending on whether dot matches newline), there is substantial extra overhead as the engine backtracks, "unmatching" characters until closing in on the ending quote from the right. At each step of the way, backtracking must ensure that it maintains the idea of "last character matched by the dot" because that's what `$1` must hold upon successful completion. The overhead can be substantial because the engine's idea of just which character that should be changes with each backtrack. Since ⌈`"(.)*"`⌋ will certainly backtrack from the end of the string to the final quote, that's potentially a lot of extra-heavy backtracking.

Example *#3* involves much less overhead than *#4*, although perhaps somewhat more than *#2*, at least in cases when there is a match or partial match. As mentioned above, *#3* has no parenthesis-related overhead until the initial quote matches. Once the ⌈.*⌋ finishes, the parentheses are exited to attempt the final quote. Its failure causes backtracking to return inside the parentheses. This happens at each step of the way until the match has been backtracked enough to match the closing quote, but it seems that the end-marker overhead of this example is much less strenuous than the full-parentheses overhead of *#4*.

I benchmarked a few examples with Tcl (Version 7.4), Python (Version 1.4bl), and Perl (Version 5.003). Figure 5-6 shows the raw timings against some rather long strings. The strings begin and end with doublequotes and have no quotes between—combined with their length, these factors help mitigate overhead unrelated to the central ⌈.*⌋ test. The Perl "non-capturing" and "non-greedy" cases refer to regexes using Perl parentheses which provide grouping only (and hence don't have the capturing-text overhead), and to regexes using Perl's non-greedy version of star.

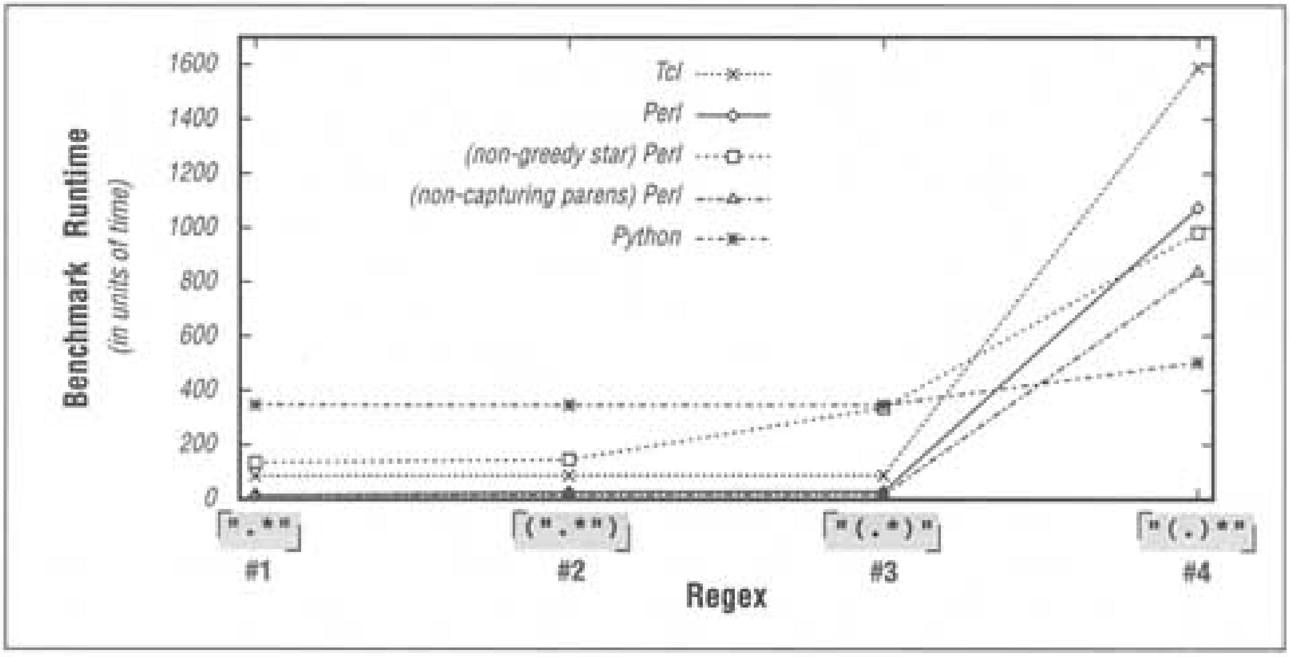


Figure 5-6:
A few parenthetical benchmarks

For the most part, the results fit the expectations I've raised. There is little relative change among the first three expressions, except in the case of Perl's non-greedy star, which must exit the parentheses at each iteration inside the long string (to see whether what follows can match). Also as expected, the times skyrocket for the ⌈`"(.)*"`⌋ case. It does this even for the Perl non-capturing case, which might seem odd at first. Theoretically, non-capturing, grouping-only parentheses add no meaning to such an expression, so they should have no influence.

The explanation lies in another reason for *#4's* relatively poor performance: the *simple-repetition* optimization discussed in the next section. Most NFA engines, Perl's included, optimize cases where quantifiers govern something "simple," such that each iteration doesn't have to work its way through the normal engine test cycle. With ⌈`(.)*`⌋, the intervening parentheses create a relatively "non-simple" situation, so the optimization is disabled. Python does not do this optimization in the first place, so it is uniformly slow. Python's time increase in *#4* seems due only to the added parenthetical overhead.

Figure 5-6 clearly illustrates the speed of *#4* relative to the others. Each implementation's relative performance of the other examples is not so clear, so I've prepared another view of the same data, Figure 5-7. Each program's data (i.e., plotted line) has been individually normalized to its own *#1's* time. In Figure 5-7, comparing the different lines for any one regex is meaningless because each line has been normalized independently. For example, the Python line is lowest, but that means nothing about its speed relative to the other lines—merely that it is the most uniform across the three expressions (most uniformly fast, or, as the case may be, uniformly slow).

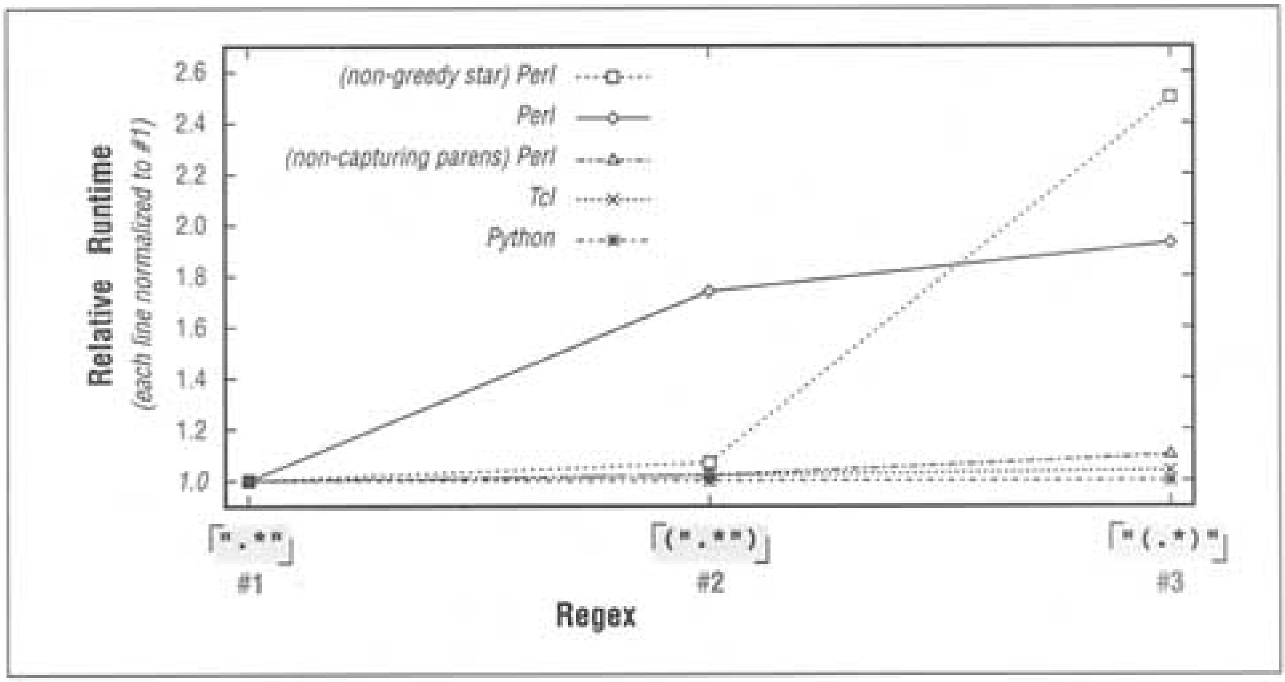


Figure 5-7:
Degrees of variation

Figure 5-7 also coincides with what we might have expected. There are slight increases for most *#3s*, with Perl's non-greedy star having a greater increase because it must exit the parentheses at each iteration of the star. One mystery at this point is why Perl's *#2* is so much slower than its *#1*. The explanation is very specific to Perl and is not exactly related to the regex engine itself. In short, it has to do with Perl's support for `$1` in the ⌈`(".*")`⌋ case. This is discussed in detail in Chapter 7 (☞ 276).

One moral to learn from all this is that it can be beneficial to reduce parentheses, or to engineer their exact location. If you use a repetition quantifier, try to put parentheses outside, or even remove them if possible. If you really need to match ⌈`".*"`⌋, but want to capture the last character of the string, as with ⌈`"(.)*"`⌋, it's much more efficient to use ⌈`(".*")`⌋ and then manually extract the next-to-last character of that `$1` (the last being the closing quote) with `substr` or the like.

Internal Optimizations

A regex engine's work is divided into two phases: analyze the regular expression while converting it to some internal form, and then check a target string against that internal form. When the same expression is used repeatedly, such as to check successive lines of a file, the analyze-and-convert phase need not be done each time. It can be done once the first time the expression is used, leaving the internal form to be used each successive time. Thus, a bit of extra effort, spent once to analyze the expression and build an internal form which can match more quickly, can pay large dividends in the end. This example is the *compile-caching* optimization discussed on page 158.

Some common optimizations are listed here. Let me stress that many optimizations can't be counted on, so it is best to program defensively for important applications. And when it really counts, benchmark.

First-Character Discrimination

Consider the 「`(Jan|Feb|…|Nov|Dec)?(31|[123]0|[012]?[1-9])`」 example from page 150. At every position where the expression is attempted, a fair amount of backtracking is required just to realize that the match fails at the first character.

Rather than paying this penalty each time, if the pre-use analyze phase realizes that any match must begin with only certain characters (in this case, a character represented by 「`[JFMASOND0-9]`」), the transmission can then quickly scan the string for such a character, handing control to the full engine only when one is found. Finding a possible-start character doesn't mean that there is a match, but merely that one is plausible any attempt at a string starting with anything else is fruitless, so such start positions are quickly skipped.

As I noted, this optimization is really part of the transmission that controls the locations in the string where match attempts begin, so it's applicable to any type of engine. The way a DFA does its pre-use compile makes its first character discrimination perfect. An NFA on the other hand, requires extra effort to create the list of possible start characters extra work that few engines do fully. Perl and Tcl, for example, make only halfhearted efforts even with something as simple as 「`am|pm`」, they are not able to determine that 「`[ap]`」 is the start list.

GNU Emacs performs very few of the optimizations mentioned in this chapter, but it does first-character discrimination, and it does it *very* well (the saving grace that makes the widespread regular expression use in Emacs reasonably possible). Even though Perl's optimizations in this area are not as good, Perl lets you do it manually. This was mentioned briefly, in the footnote on page 149, and is discussed in more detail in Chapter 7, starting on page 228.

Of course, there are many expressions for which no first-character discrimination is possible, such as with any expression beginning with ⌈`.`⌋ (since a match could begin at any character, or at least any except newline or null in some flavors).

Fixed-String Check

If the pre-use compile-time analysis reveals that some particular fixed string must exist in any possible match, such as the `'Subject:•'` of ⌈`^Subject:•(Re:•)?(.*)`⌋, or even the quote of ⌈`".*"`⌋, the transmission might employ a different technology to quickly rule out targets lacking the fixed string. The idea is that because the other technology is so much faster than the general-purpose regex engine, spending the additional time to do the pre-check can save time in the end. Usually, an algorithm called "Boyer-Moore" us used.

As with the previous optimization, a DFA's analysis intrinsically supports a fixedstring check very well, but NFA engines are 'not usually very thorough. We can glance at ⌈**`this|that|them`**⌋, for example, and realize that **`th`** is required in the target string for there to be any possibility of a match, but most NFA engines won't—to the engine, a quick peek reveals only that "there is alternation." Most NFA engines won't look deeper, and so just give up on this optimization.

Note, though, that if we manually change the regex to ⌈`th(is|at|em)`⌋, an NFA engine will see a top level "fixed string '`th`', followed by alternation," which is more amenable to this (and the previous) optimization.

Perl can report when this kind of optimization is possible. If your version has been compiled with internal debugging support, you can use the `-Dr` command-line option (`-D512` with ancient versions of Perl) to have Perl report information about each regex (☞ 285). With the above example, it reports (among other things) "'`start 'th' minlen 2`'." The `minlen` information relates to *length cognizance,* discussed shortly.

Somewhat related to a fixed-string check, Perl attempts an interesting optimization when you use its study function (☞ 287). It spends a fair amount of time and a lot of memory to analyze the (perhaps long) string so that later, when the string is the target of a search, the engine can *immediately* know whether there are any, say, quotes at all. If not, it could omit checking, say, ⌈`".*"`⌋ entirely.

Simple Repetition

Uses of plus and friends that apply to simple items, such as literal characters and character classes, are often optimized such that much of the step-by-step overhead of a normal NFA match is removed. (This optimization is not applicable to the text-directed nature of a DFA engine.) I'll compare the normal overhead of a match to how an internal combustion engine works: turning the crankshaft involves mixing gasoline and air, spraying the combination into the piston's cylinder, compressing it as the piston pumps upwards, igniting the spark plug at just the right moment to cause a small explosion that forces the piston down, and then opening the valves to clear out the exhaust.

Each task is done from scratch with every cycle of the engine, but some steps become more efficient if the engine is supercharged (by using superheated air to create more efficient mini explosions). The analogy may be superficial at best, but an NFA engine can supercharge how it deals with a quantifier applied to a very simple subexpression. The main control loop inside a regex engine must be general enough to deal with all the constructs the engine supports, and in programming, "general" often means "slow."

In specific cases such as ⌜`x*`⌟, ⌜`[a-f]+`⌟, ⌜`.?`⌟, and the like, a special mini-engine that handles the few special cases more quickly can short-circuit the general "must deal with everything" engine. This optimization is quite common, and usually quite substantial. For example, benchmarking often shows that ⌜`.*`⌟ is a fair amount faster than ⌜`(.)*`⌟, due both to this optimization and to the lack of parentheses overhead. In this case, parentheses provide a double-whammy parentheses are "non-simple," so they disable the simple-repetition optimization and add their own capturing-related overhead as well.

Figure 5-8 shows the same data as Figure 5-7, except the ⌜`"(.)*"`⌟case has been included. In other words, it's the same as Figure 5-6 but each line is independently normalized to regex *#1*. Figure 5-8 tells us a lot about the simple-repetition optimization. With ⌜`"(.)*"`⌟, Perl's non-capturing parentheses should (theoretically) have no influence, but in practice it appears that their presence blocks Perl from noticing that it can invoke this optimization, resulting in a 50-fold slowdown. Perl's normal capturing parentheses suffer the same fate, so the difference between the two (16 units on the graph) remains as the cost of the parentheses overhead.

Finally, how do we explain the results of Perl's non-greedy star? Well, since the engine must continually leave the non-greedy ⌈`.*`⌋ to see whether what follows can match, this simple-repetition operator can't really be invoked. So, *#3* also has the per-character parentheses overhead, which explains why the relative slowdown for Perl's non-greedy star, from *#3* to *#4*, isn't so great. When you're slow to begin with, being slow in a slightly different way is easy.

Needless Small Quantifiers

Similarly, something like ⌈`xxx`⌋ is probably much faster than ⌈`x{3}`⌋. The `{count}` notation is useful, but when it's applied to a small number of a simple item, you can relieve the engine from having to count occurrences by listing the desired items explicitly.

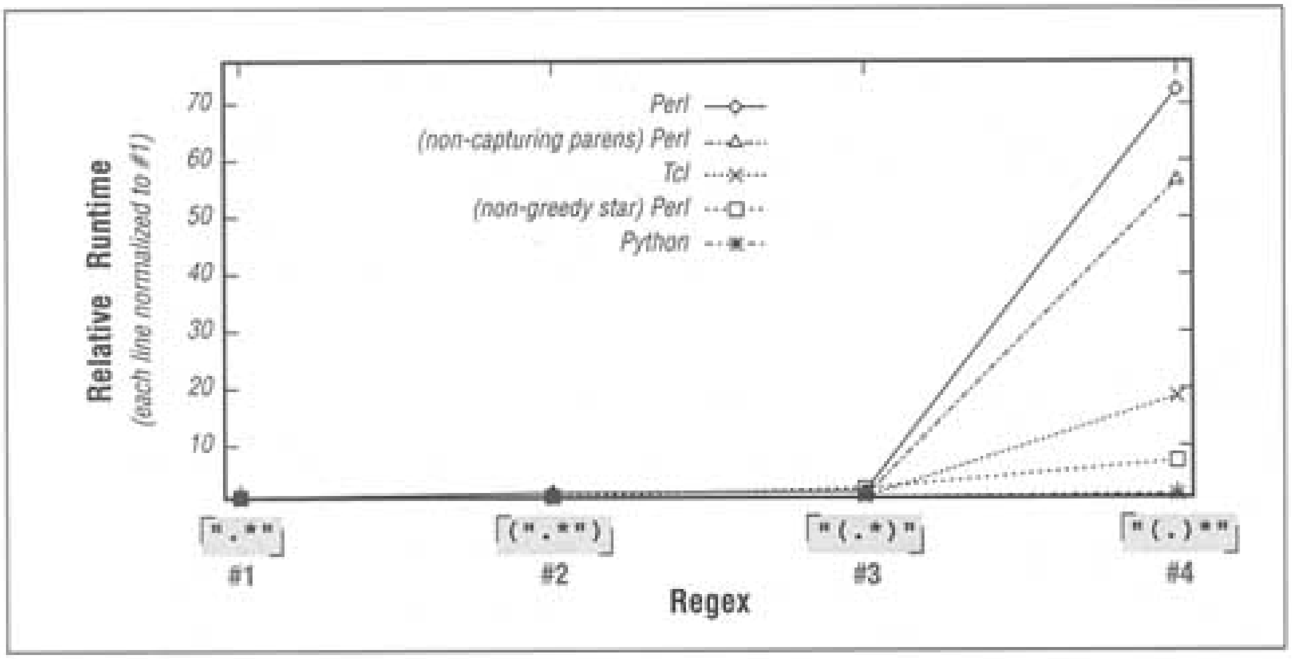


Figure 5-8:
Degrees of variation (full data)

Length Cognizance

It's a small optimization, but if the compile-time analysis reveals that any match must be at least a particular length, shorter strings can be safely discarded. Similarly, match attempts that might start within that distance from the end of the string can be skipped. As with the other optimizations that require deep analysis of the entire regex, DFAs can do it quite well, while NFAs often skimp.

Match Cognizance

If a POSIX NFA finds a match that continues all the way to the end of the string, it's obvious that a longer match is impossible so there's no need to bother searching any further. Due to greediness, matches that can continue to the end of the string often do so rather early during the matching   this optimization can yield huge savings in such cases.

Need Cognizance

If a regex is used in a situation where the exact extent of the match is not a concern (such as in a pure "if it matches" situation), the engine can stop the moment it finds any match at all. This optimization is of primary interest to DFA and POSIX NFAs. *egrep,* for example, never cares about which text on the line matches   only whether there is or isn't a match. GNU *grep* realizes this, so its DFA does a *shortest*-leftmost match instead of wasting time with a full longest-leftmost match. Similarly, Michael Brennan's *awk* (*mawk*) uses a POSIX NFA, but reverts to a Traditional NFA when the exact extent of the match is not required.

String/Line Anchors

A very simple optimization is realizing that if the regex (or every alternative) begins with a caret, the match must start at the beginning of the string or not at all, so the transmission's bump-alongs can be eliminated. If caret can match after an embedded newline (☞ 81), a bump-along *is* required, but the transmission can quickly scan to the next newline, eliminating all the (possibly many) tests in between.

Implicit line anchor

A related optimization realizes that any regex beginning with 「`.*`」 that doesn't match at the start of the string certainly can't match starting at any later position. In that case, it's safe to prepend 「`^`」 internally. We saw this effect in the pathname examples on page 132. Trying to match 「`.*/`」 against '`some.long.filename`', the initial 「`.*`」 matches to the end of the string before backtracking in vain as it tries to match the slash. The attempt from the start of the string thus fails, but without an anchor, the transmission then continues to apply the regex at subsequent starting positions. Since the leading 「`.*`」 has *already*, in effect, applied the rest of the regex (only a slash, in this case) to each position in the string, there is no hope left to find a match—any subsequent attempts are an utterly wasted effort.

Well, there's still the "dot can't match newline" issue. If there are characters that dot can't match, a match could indeed begin after a such a character (commonly newline or null). This would be similar to where line anchors could match after an embedded newline. If a match isn't found starting at the beginning of the string, the transmission can still optimize by starting subsequent attempts only after each can't-match-dot character.

Compile Caching

As mentioned briefly on page 118, and at the start of this section, a regex is compiled to an internal form before actually being applied for searching text. The compile takes some time, but once done, the result can be applied any number of times. For example, *grep* compiles the regex just once, and then applies it to all lines of all files being checked.

The compiled form *can* be used multiple times, but *will it?* Regular expressions in languages like *awk*, GNU Emacs, Perl, and so on, are usually used in unpredictable ways, so it's not necessarily a straightforward task to maintain the compile-once, use-over paradigm. Consider something as simple as *grep*-like routines in Tcl and Perl to print lines of a file that match ⌈`[Tt]ubby`⌋ (shown on the next page).

Tcl —

```
while {[gets $file line]!=—l} {
   if {[regexp {[Tt]ubby} $line]} {
         puts $line
   }
}
```

Perl —

```
while (defined($line = <FILE>))
    if ($line =~ /[Tt]ubby/)
         print $line;
    }
}
```

In the examples above, the regex is potentially compiled and used (used just once) with *each iteration* of the while loop. *We* know, by looking at the overall logic of the program, that it's the same regex being applied each time, so reusing the same compiled form over and over would save a lot of recompilation (which saves a lot of time). Unfortunately, Tcl and Perl don't necessarily know that. In theory, they must compile the regex afresh with each use. In practice, though, there are methods to get around some of the work. Let's compare how Perl and Tcl provide their match features.

Functions vs. integrated features vs. objects

Tcl's match is a normal function, `regexp`. As a function, it knows nothing about where its arguments came from. The `{…}` notation is Tcl's non-interpolative, truly literal string, so we can look at `{[Tt]ubby}` and know it can never change from use to use. But `regexp` merely sees the '`[Tt]ubby`' that it eventually receives—it doesn't know if the argument originally came from a literal string, an interpolated string, a variable, or what.

Compare this to Perl, whose match is an operator. An operator can understand more about itself and its operands than a function can know about how it is called or about how its arguments are provided. In the example, the match operator *knows* that ⌈`[Tt]ubby`⌋ is a raw regex that will never change from use to use, so the compiled form is saved and reused over and over. This is a huge savings. Yet, had the regex been in a variable, say `$regex`, interpolated into the match (that is, used as **`$line =~ /$regex/`),** Perl would know that the regex *could* change from use to use (depending each time on the value of `$regex`)

Even in this example, we can see `$regex` won't change during the course of the loop, but Perl doesn't have that high-level understanding. However, since the match operator is not a generic function, it can know *which* match operator, among all in the program, it is, and can remember what regex was used the last time it was invoked. It does a simple string compare between the old and new regex (after any variable interpolation), and reuses the compiled form if they're the same. If they're different, the entire regex is recompiled.

Since Tcl's `regexp` function doesn't know anything about where its regex argument comes from, or about from where in the script it was called, you might think that it has to fully recompile each time, but Tcl is not without its bag of tricks. Tcl keeps a cache of the internal forms of the five most recently used regular expressions in the entire script. The first time through the loop, `regexp` compiles the regular expression and uses it to perform the match. Each subsequent iteration of the loop, however, gives regexp the same regex, so it's found in the cache. The cache must be searched each time, but bypassing the compile is still a big savings.

Even with Perl's smarts about compiling, a programmer sometimes knows that the "Is this the same as before?" check is unnecessary. To help stem the inefficiency, Perl provides methods to put some of the control into the programmer's hands. See "Perl Efficiency Issues" in Chapter 7 (☞ 268).

Further along the same lines, Python allows the programmer to take *complete* control. Python supports normal "use me now" regexes in a way similar to Tcl's `regexp`, but it also supports compiled regular-expressions as first-class objects (☞ 69). This allows the programmer to separate a regex's compile from its use. In our example, the regex would be compiled once, before the loop. The resulting *compiled-regex object* can then be used within the loop:

```
    CompiledRegex = regex.compile("[Tt]ubby");   # Compile and save
regex
    while 1:                                     # For entirefile. . .
        line = file.readline()                   #     read line
        if not line: break                       #     if nothing, we're
done
        if (CompiledRegex.search(line) >= 0):    #     Apply compiled
regex to line. . .
            print line,                          #     . . .and print if a
match.
```

This can be more work for the programmer, but it affords the most control. When used skillfully, that translates into efficiency. An example on page 177 illustrates how important understanding these issues can be. Like Tcl, GNU Emacs caches five regexes, but changing that number to 20 resulted in an almost threefold speedup for the Emacs version of that example.

Testing the Engine Type

Testing a tool's regex engine can be a multi-step process. The first step is determining whether it uses an NFA or DFA. If it uses an NFA, the next step is to see whether it is POSIX or not.

Basic NFA vs. DFA Testing

In theory, testing for the basic engine type is as simple as testing for a neverending match. In practice, some tools use the optimizations just discussed to bypass a test altogether (giving the appearance of not being a neverending match, and thus giving the appearance of being a DFA). Here's a way to test *egrep* that gets around all optimizations I'm aware of:

```
    echo =XX================================================= |
egrep "X(.+)+X"
```

The doublequotes are for the shell; the regex is ⌜`X(.+)+X`⌟. The match should fail: if it does so immediately, the engine is a DFA.* If it takes more than a few seconds, it's an NFA (and you should stop the program manually because it probably won't finish in this lifetime). You can also try ⌜`X(.+)*X`⌟ (which should match) to see whether the match is reported immediately, or takes forever.

You can apply this kind of test to any engine you like, although you might have to adjust the regex flavor accordingly. For example, with GNU Emacs, you could put the `=XX===…` line in the buffer and use `isearch-forward-regexp` (default keybinding: `M-C-s`) to search for ⌜`X\(.+\)+X`⌟. (GNU Emacs uses ⌜`\(…\)`⌟ for grouping.) It's an interactive search, so the moment you type the second `x`, it "locks up" until you abort with `C-g`, or until the engine eventually works through the match. This shows GNU Emacs uses an NFA. On the other hand, you might test *awk* with:

```
echo =XX==…=== | awk "/[X(.+)*X/{print}"
```

If it immediately prints the `=xx==…` line, your *awk* uses a DFA (as most do). If, for example, your *awk* is *mawk* or Mortice Kern Systems's *awk*, you'll find that it uses an NFA.

Traditional NFA vs. POSIX NFA Testing

Since a Traditional NFA can stop the moment it finds a match, these neverending regexes often finish quickly when a match is possible. Here's a GNU Emacs example: create a `=XX=================`**`X`** line (note that a trailing `X` as been added), and apply ⌜`X\(.+\)+X`⌟ again. You'll find this time that it has no trouble matching immediately. Now try the same regex with `posix-search-forward`, the POSIX-engine search function, and you'll find it "locks up."

Traditional versus POSIX testing is often made difficult by the optimizations I've discussed. If you try the *awk* example with the appended `X` and use **mawk** (which uses a POSIX NFA), you'll find that it returns the successful match immediately even though it is POSIX and (theoretically) must attempt the same gazillion permutations as when it can't match. The reason it returns immediately is because it employs a *need-cognizance optimization*—the binary "does it match or not" use of the regex means that the *extent* of the match isn't important, so there's no need to bother trying to find the longest match (or any particular match, for that matter).

On the other hand, using the same regex where the exact match is important, **mawk** shows its POSIXness and locks up:

```
echo =XX===…===X | mawk "{sub(/X(.+)*X/, replacement)}"
```

---

* Of course, it could also be an NFA with an optimization that I have not foreseen.

If the engine had the *match-cognizance* optimization, it would be able to stop immediately even with this test, since *a* match to the end of the string would come up right away, and the match-cognizance optimization would realize that a longer match is not possible. In this case, the testing needs a bit more ingenuity. It could be as simple as adding '===' to the end of the test string to let the engine think a longer match might be possible if it were to just keep looking.

Unrolling the Loop

Now that we've reviewed the basics in excruciating detail, let's step up to the big leagues. A technique I call "unrolling the loop" is effective for speeding up certain common expressions. The loop in question is via the star in an expression that fits a ⌜`(this|that|…)*`⌟ pattern. You might recognize that our earlier neverending match, ⌜`"(\\.|[^"\\]+)*"`⌟, fits this pattern. Considering that it will take approximately forever to report a non-match, it's a good example to try to speed up!

There are two competing roads one can take to arrive at this technique:

1. We can examine which parts of ⌜`(\\.|[^"\\]+)*`⌟ actually succeed during a variety of sample matches, leaving a trail of used subexpressions in its wake. We can then reconstruct an efficient expression based upon the patterns we see emerge. The (perhaps far-fetched) mental image I have is that of a big ball, representing a ⌜`(…)*`⌟ regex, being rolled over some text. The parts inside `(…)` that are actually used then stick to the text they match, leaving a trail of subexpressions behind like a dirty ball rolling across the carpet.

2. Another approach takes a higher-level look at the construct we want to match. We'll make an informed assumption about the likely target strings, allowing us to take advantage of what we believe will be the common situation. Using this point of view, we can construct an efficient expression.

Either way, the resulting expressions are identical. I'll begin from the "unrolling" point of view, and then converge on the same result from the higher-level view.

Method 1: Building a Regex From Past Experiences

In analyzing ⌈`"(\\.|[^"\\]+)*"`⌋, it's instructive to look at some matching strings to see exactly which subexpressions are used during the overall match. For example, with '`"hi"`', the expression effectively used is just ⌈`"[^\\]+"`⌋. This illustrates that the overall match used the initial ⌈`"`⌋, one application of the alternative ⌈`[^"\\]+`⌋, and the closing ⌈`"`⌋. With

```
"he said \"hi there\" and left"
```

it is ⌈`"[^\\]+\\.[^"\\]+\\.[^"\\]+"`⌋. In this examples, as well as in Table 5-2, I've marked the expressions to make the patterns apparent. It would be nice if we could construct a specific regex for each particular input string. That's possible, but we can still identify common patterns to construct a more-efficient, yet still general, regular expression.

Table 5-2: Unrolling-The-Loop Example Cases

Target String	**Effective Expression**
`"hi there"`	`"[^"\\]+"`
`"just one \" here"`	`"[^"\\]+\\.[^''\\]+"`
`"some \"quoted\" things"`	`"[^"\\]+\\.[^"\\]+\\"[^"\\]+"`
`"with \"a\" and \"b\"."`	`"[^"\\]+\\.[^"\\]+\\.[^"\\]+"\\.[^"\\]+"\\.[^"\\]+"`
`"\"ok\"\n"`	`"\\.[^"\\]+\\.\\."`
`"empty \"\" quote"`	`"[^"\\]+\\.\\.[^"\\]+"`

Table 5-2 shows a few more examples. For the moment, let's concentrate on the first four. I've underlined the portions that refer to "escaped item, followed by further normal characters." This is the key point: in each case, the expression between the quotes begins with ⌈`[^"\\]+`⌋ and is then followed by some number of ⌈`\\.[^"\\]+`⌋ sequences. Rephrasing this as a regular expression, we get ⌈`[^"\\]+(\\.[^"\\]+)*`⌋. This is a specific example of a general pattern that can be used for constructing many useful expressions.

Constructing a general "unrolling the loop" pattern

In matching the doublequoted string, the quote itself and the escape are "special"   the quote because it can end the string, and the escape because it means that whatever follows won't end the string. Everything else, ⌈`[^"\\]`⌋, is "normal." Looking at how these were combined to create ⌈`[^"\\]+(\\.[^"\\]+)*`⌋ we can see that it fits the general pattern ⌈*normal*`+ (`*special normal*`+)*`⌋.

Adding in the opening and closing quote, we get ⌈`"[^"\\]+(\\.[^"\\]+)*"`⌋. Unfortunately, this won't match the last two examples in Table 5-2. The problem, essentially, is that this expression's two ⌈`[^"\\]+`⌋ *require* a normal character at the start of the string and after any special character. As the examples show, that's not always appropriate   the string might start or end with an escaped item, or there might be two escaped items in a row.

We could try changing the two pluses to stars: ⌈`"[^"\\]*(\\.[^"\\]*)*"`⌋. Does this have the desired effect? More importantly, does it have any undesirable effects?

As far as desirable effects, it is easy to see that all the examples now match. In fact, even a string such as `"\"\"\""` now matches. This is good. However, we can't make such a major change without being quite sure there are no undesirable effects. Could anything other than a legal doublequoted string match? Can a legal doublequoted string not match? What about efficiency?

Let's look at ⌜`"[^"\\]*(\\.[^"\\]*)*"`⌟ carefully. The leading ⌜`"[^"\\]*`⌟ is matched only once and seems harmless: it matches the required opening quote and any normal characters that might follow. No danger here. The subsequent ⌜`(\\.[^"\\]*)*`⌟ is wrapped by **`(…)*`**, so is allowed to match zero times. That means that removing it should still leave a valid expression. Doing so, we get ⌜`"[^"\\]*"`⌟, which is certainly fine—it represents the common situation where there are no escaped items.

On the other hand, if ⌜`(\\.[^"\\]*)*`⌟ matches once, we have an effective ⌜`"[^\\]*\\.[^"\\]*"`⌟. Even if the trailing ⌜`[^"\\]*`⌟ matches nothing (making it an effective ⌜`"[^\\]*\\."`⌟), there are no problems. Continuing the analysis in a similar way (if I can remember my high school algebra, it's "by induction"), we find that there are, indeed, no problems with the proposed changes.

The Real "Unrolling the Loop" Pattern

Putting it all together, then, our expression to match a doublequoted string with escaped-items is ⌜`"[^\\]*(\\.[^"\\]*)*"`⌟ This matches exactly the same strings as our alternation version, and it fails on the same strings that the alternation version fails on. But this unrolled version has the added benefit of finishing in our lifetime because it is much more efficient and avoids the neverending-match problem.

The general pattern for this kind of expression is:

⌜`opening normal*(special normal*)*closing`⌟

Avoiding the neverending match

Three extremely important points prevent ⌜`"[^"\\]*(\\.[^"\\]*)*"`⌟ from becoming a neverending match:

The start of special and normal must never intersect

First, the *special* and *normal* subexpressions must be written such that they can never match at the same point. For example, ⌈`\\.`⌋ and ⌈`[^"]`⌋ can both match starting at ‘`"Hello\n"`’, so they are inappropriate *special* and *normal*. If there is a way they can match starting at the same location, it's not clear which should be used at such a point, and the non-determinism creates a neverending match. The ‘`makudonarudo`’ example (☞ 144) illustrated this graphically. A failing match (or any kind of match attempt with POSIX NFA engines) would have to test all these possibilities and permutations. That would be a bummer, since the whole reason to re-engineer in the first place was to avoid this.

If we ensure that *special* and *normal* can never match at the same point, *special* acts to checkpoint the nondeterminism that would arise when multiple applications of *normal* could, by different iterations of the ⌈`(...)*`⌋ loop, match the same text. If we ensure that *special* and *normal* can never match at the same point, there will be exactly one possible "sequence" of *specials* and *normals* in which a particular target string could match. Testing the one sequence is much faster than testing a hundred million of them, and thus a neverending match is avoided.

We've satisfied this rule in our ongoing example, where *normal* is ⌈`[^"\\]`⌋ and *special* is ⌈`\\.`⌋. They can never begin a match at the same character   the latter requires a leading backslash, while the former explicitly disallows one.

Special must not match nothingness

The second important point is that *special* must always match at least one character if it matches anything at all. If it could match without consuming characters, adjacent normal characters would be able to be matched by different iterations of ⌈**(***special normal****)***⌋, bringing us right back to the basic `(…*)*` problem.

For example, choosing a *special* of ⌈`(\\.)*`⌋ violates this point. In trying to match the ill-fated* ⌈`"[^"\\]*((\\.)*[^"\\*)*"`⌋ against '`"Tubby`' (which doesn't match), the engine must try every permutation of how multiple ⌈`[^"\\]*`⌋ might match `'Tubby'` before concluding that the match is a failure. Since *special* can match nothingness, it doesn't act as the checkpoint it purports to be.

Text matched by one application of special must not be able to be matched by multiple applications of special

The third important point is best shown by example. Consider matching a string of optional Pascal `{…}` comments and spaces. A regex to match the comment part is ⌈**`\{`**`[^}]*`**`\}`**⌋, so the whole (neverending) expression becomes ⌈**(**`\{[^}]*\}`**|**`•+`**)**`*`⌋. With this regex, you might consider *special* and *normal* to be:

special	normal

⌈`•+`⌋   ⌈`\{[^}]*\}`⌋

Plugging this into the ⌈`normal* (special normal*)*`⌋ pattern we've developed, we get: ⌈`(\{[^}]*\})*( •+(\{[^}]*\})* )*`⌋. Now, let's look at a string:

```
{comment}•••{another}••
```

A sequence of multiple spaces could be matched by a single ⌈`•+`⌋, by many ⌈`•+`⌋ (each matching a single space), or by various combinations of ⌈`•+`⌋ matching differing numbers of spaces. This is directly analogous to our 'makudonarudo' problem.

The root of the problem is that *special* is able to match a smaller amount of text *within* a larger amount that it could also match, and is able to do so multiple times via `(…)*`. The nondeterminism opens up the "many ways to match the same text" can of worms.

---

* Many NFA engines disallow the ⌈`(x*y*)*`⌋ pattern seen in this regular expression. Since the inner subexpression can match nothingness, the outer star could "immediately" apply it an infinite number of times. Other tools, such as Emacs and modern versions of Perl, handle the situation with grace. Python quietly fails on all matches that involve such a construct.

If there is an overall match, it is likely that only the all-at-once, just-once ⌈•+⌋ will happen, but if no match is possible (such as becomes possible if this is used as a subexpression of a larger regex that could possibly fail), the engine must work through each permutation of the effective ⌈(•+)*⌋ to each series of multiple spaces. That takes time, but without any hope for a match. Since *special* is supposed to act as the checkpoint, there is nothing to check *its* nondeterminism in this situation.

The solution is to ensure that *special* can match only a fixed number of spaces. Since it must match at least one, but could match more, we simply choose ⌈•⌋ and let multiple applications of *special* match multiple spaces via the enclosing ⌈(…)*⌋.

This example is useful for discussion, but if I actually wanted this kind of regex, I would swap the *normal* and *special* expressions to come up with

⌈•*(\{[^}]*\}•*)*⌋

because I would suspect that a Pascal program has more spaces than comments, and it's more efficient to have *normal* be the normal case.

General things to look out for

Once you internalize these rules (which might take several readings and some practical experience), you can generalize them into guidelines to help identify regular expressions susceptible to a neverending match. Having multiple levels of quantifiers, such as ⌈(…*)*⌋, is an important warning sign, but many such expressions are perfectly valid. Examples include:

- ⌈(Re:•*)*⌋, to match any number of '`Re:`' sequences (such as might be used to clean up a '`Subject:•Re:•Re:•Re:•hey`' subject line).

- ⌈`(·*\$[0-9]+)*`⌋, to match space-separated dollar amounts.

- ⌈`(.*\n)+`⌋, to match one or more lines. Actually, if dot can match a newline, and if there is anything following this subexpression that could cause it to fail, this would become a quintessential neverending match.

These are okay because each has something to checkmark the match, keeping a lid on the "many ways to match the same text" can of worms. In the first, it's ⌈`Re:`⌋, in the second it's ⌈`\$`⌋, and in the third (when dot doesn't match newline), it's ⌈`\n`⌋.

Method 2: A Top-Down View

Recall that I said that there were two paths to the same expression. Let's consider what the neverending ⌈`(\\.|[^"\\]+)*`⌋ attempts to accomplish and where it will likely be used. Normally, I would think, a quoted string would have more regular characters than escaped items, so ⌈`[^"\\]+`⌋ does the bulk of the work. The ⌈`\\.`⌋ is needed only to take care of the occasional escaped item. Using alternation to allow either makes a useful regex, but it's too bad that we need to compromise the efficiency of the whole match for the sake of a few (or more commonly, no) escaped characters.

If we think that ⌈`[^"\\]+`⌋ will normally match most of the body of the string, then we know that once it finishes, we can expect either the closing quote or an escaped item. If we have an escape, we want to allow one more character (whatever it might be), and then match more of the bulk with another ⌈`[^"\\]+`⌋. Every time ⌈`[^"\\]+`⌋ ends, we are in the same position we were before: expecting either the closing quote or another escape.

Expressing this naturally as a single expression, we arrive at the same expression we had early in Method 1: ⌈`"[^"\\]+(↱\\.[^"\\]+)*"`⌋. Each time the matching reaches the point marked by ↱, we know that we're expecting either a backslash or a closing quote. If the backslash can match, we take it, the character that follows, and more text until the next "expecting a quote or backslash" point.

As in the previous method, we need to allow for when the initial non-quote segment, or inter-quote segments, are empty. We can do this by changing the two pluses to stars, which results in the same expression as the other two methods.

Method 3: A Quoted Internet Hostname

I promised two methods to arrive at the *unrolling-the-loop* technique, but I'd like to present a similar method that can be considered a third. It struck me while working with a regex to match a domain name such as `prez.whitehouse.gov` or `www.yahoo.com`—essentially period-separated lists of subdomain names. For the purposes of the example, we'll consider the simple (but incomplete) ⌈`[a-z]+`⌋ to match a subdomain.

If a subdomain is ⌜`[a-z]+`⌟ and we want a period-separated list of them, we need to match one subdomain first. After that, any further subdomains are optional, but require a leading period. Expressing this literally, we get: ⌜`[a-z]+(\.[a-z]+)*`⌟. Especially if I write it as ⌜`[a-z]+(\.[a-z]+)*`⌟, it sure looks like it almost fits a very familiar pattern, doesn't it!

To illustrate the similarity, let's try to map this to our doublequoted string example. If we consider a string to be sequences of our *normal* ⌜`[^\\"]`⌟, separated by *special* ⌜`\\.`⌟, all within '`"…"`', we can plug them into our unrolling-the-loop pattern to form ⌜`"[^\\"]+(\\.[^\\"]+)*"`⌟, which is exactly what we had at one point while discussing Method 1. Considering the contents of a doublequoted string to be "sequences of non-escaped stuff separated by escaped items" isn't exactly natural, but yields an interesting path to what we've already seen.

There are two differences between this and the subdomain example:

- Domain names have no wrapping delimiters.

- The *normal* part of a subdomain can never be empty (meaning two periods are not allowed in a row, and can neither start nor end the match). With a doublequoted string, there is no requirement that there be any *normal* parts at all, even though they are likely, given our assumptions about the data. That's why we were able to change the ⌜`[^\\"]+`⌟ to ⌜`[^\\"]*`⌟. We can't do that with the subdomain example because *special* represents a separator, which is required.

Observations

Recapping the doublequoted-string example, I see many benefits to our expression, ⌜`"[^\\]*(\\.[^\\]*)*"`⌟, and few pitfalls.

Pitfalls:

- **readability** The biggest pitfall is that the original ⌜`"([^"\\]|\\.)*"`⌟ is probably easier to understand at first glance. We've traded a bit of readability for efficiency.

- **maintainability** Maintaining ⌜`"[^"\\]*(\\.[^"\\]*)*"`⌟ might be more difficult, since the two copies of ⌜`[^"\\]`⌟ must be kept identical across any changes. We've traded a bit of maintainability for efficiency.

Benefits:

- **speed** The new regex doesn't buckle under when no match is possible (or when used with a POSIX NFA). By carefully crafting the expression to allow only one way for any particular span of text to be matched, the engine can quickly come to the conclusion that non-matching text indeed does not match.

- **more speed** The regex "flows" well, a subject taken up in "The Freeflowing Regex" (☞ 173). In my benchmarks with a Traditional NFA, the unrolled version is consistently faster than the old alternation version. This is true even for successful matches, where the old version did not suffer the lockup problem.

Unrolling C Comments

I'd like to give an example of unrolling the loop with a somewhat more complex target. In the C language, comments begin with `/*`, end with `*/`, and can span across lines (but can't be nested). An expression to match such a comment might be useful in a variety of situations, such as in constructing a filter to remove them. It was when working on this problem that I first came up with my unrolling technique, and it has since become a staple in my regex arsenal.

Regex Headaches

There are no escapes recognized within a `C` comment the way an escaped quote is recognized within a doublequoted string. This should make things more simple, but matching `C` comments is much more complex because `*/`, the "ending quote," is more than one character long. The simple ⌈`/\*[^*]*\*/`⌋ won't work here because it won't match something like `/**` some comment here `**/` which validly has a '`*`' within. We need a more complex approach.

You might find that ⌈`/\*[^*]*\*/`⌋ is a bit difficult to read, even with the easy-on-the-eyes spacing that I have used in typesetting this book—it is unfortunate for our eyes that one of the comment's delimiting characters, '`*`', is also a regex metacharacter. The resulting backslashes are enough to give me a headache. To make things more readable during this example, we'll consider `/x`…`x/`, rather than `/*`…`*/`, to be our target comment. This superficial change allows ⌈`/\*[^*]*\*/`⌋ to be written as the more readable ⌈`/x[^x]*x/`⌋. As we work through the example and the expression becomes more complex, our eyes will thank us for the reprieve.

A Naive View

In Chapter 4 (☞ 129), I gave a standard formula for matching delimited text:

1. match the opening delimiter

2. match the main text: really "match anything that is not the ending delimiter"

3. match the ending delimiter

Our pseudo comments, with `/x` and `x/` as our opening and closing delimiters, appear to fit into this pattern. Our difficulties begin when we try to match "anything that is not the ending delimiter." When the ending delimiter is a single character, we can use a negated character class to match all characters except that delimiter. There is, however, no general way to say "anything that is not this multi-character delimiter,"* so we must craft the regex more carefully. There are, perhaps, two ways to go about matching until the first `x/`. One method is to consider `x` to be the start of the ending delimiter. That means we'd match anything not `x`, and allow an `x` if it is followed by something other than a slash. This makes the "anything that is not the ending delimiter" one of:

- anything that is not `x`: ⌜`[^x]`⌟
- x, so long as not followed by a slash: ⌜`x[^/]`⌟

This yields ⌜`([^x]|x[^/])*`⌟ to match the main text, and ⌜`/x([^x]|x[^/])*x/`⌟ to match the entire pseudo comment.

---

* Actually, Perl Version 5 does provide this via its `(?!...)` negative-lookahead construct, but since it is so specific to Perl, I'll leave it until Chapter 7 (☞ 228).

Another method is to consider slash to be the ending delimiter, but only if preceded by `x`. This makes the "anything not the ending delimiter" one of:

- anything that is not a slash: ⌈`[^/]`⌋
- a slash, so long as not preceded by `x`: ⌈`[^x]/`⌋

This yields ⌈`([^/]|[^x]/)*`⌋ to match the main text, and ⌈`/x([^/]|[^x]/)*x/`⌋ to match the whole comment.

Unfortunately, neither method works.

For ⌈`/x([^x]|x[^/])*x/`⌋, consider '`/xx·foo·xx/`'—after matching '`foo·`', the first closing `x` is matched by ⌈`x[^/]`⌋, which is fine. But then, ⌈`x[^/]`⌋ matches `xx/`, which is the `x` that should be ending the comment. This allows the match to continue past the closing `x/` (to the end of the next comment if one exists).

As for ⌈`/x([^/]|[^x]/)*x/`⌋, it can't match '**`/x/·foo·/x/`**' (the whole of which is a comment and should be matched). In other cases, it can march past the end of a comment that has a slash immediately after its end (in a way similar to the other method). In such a case, the backtracking involved is perhaps a bit confusing, so it should be instructive to understand why ⌈`/x([^/]|[^x]/)*x/`⌋ matches

```
    years = days /x divide x//365; /x assume non-leap year x/
```

as it does (an investigation I'll leave for your free time).

Making it work

Let's try to fix these regexes. With the first one, where 「`x[^/]`」 inadvertently matches the comment-ending …`xx/`, consider 「`/x([^x]|x+[^/])*x/`」. The added plus will, we think, have 「`x+[^/]`」 match a row of `x`'s ending with something other than a slash. Indeed it will, but due to backtracking, that "something other than a slash" can still be `x`. At first, the greedy 「`x+`」 matches that extra `x` as we want, but backtracking will reclaim it if needed to secure an overall match. Unfortunately, it still matches too much of:

```
/xx A xx/ foo() /xx B xx/
```

The solution comes back to something I've said before: *say what you mean.* If we want "some x, if not followed by a slash" to imply that the non-slash also doesn't include an x, we should write exactly that: 「`x+[^/x]`」. As we want, this stops it from eating '…`xxx/`', the final `x` of a row of `x` that ends the comment. In fact, it has the added effect of not matching *any* comment-ending `x`, so it leaves us at '…`xxx/`' to match the ending delimiter. Since the ending delimiter part had been expecting just the one `x`, it won't match until we insert 「`x+/`」 to allow this final case.

This leaves us with: 「`/x([^x]|x+[^/x])*x+/`」 to match our pseudo

Translating Between English and Regex

On page 169, when discussing two ways one might consider the C comment "anything that is not the ending delimiter," I presented one idea as

> `x`, so long as not followed by a slash: ⌈`x[^/]`⌋

and another as:

> a slash, so long as not preceded by `x`: ⌈`[^x]/`⌋

In doing so, I was being informal—the English descriptions are actually quite different from the regexes. Do you see how?

To see the difference, consider the first case with the string '`regex`'—it certainly has an `x` not followed by a slash, but it would not be matched by match ⌈`x[^/]`⌋. The character class requires a character to match, and although that character can't be a slash, it still must be something, and there's nothing after the `x` in '`regex`'. The second situation is analogous. As it turns out, what I need at that point in the discussion are those specific expressions, so it's the English that is in error.

By the way, as an aside, note that if you really did want to implement "`x`, so long as not followed by a slash" you could try ⌈`x([^/]|$)`⌋ . It still matches a character after the `x`, but can also match at the end of the line. A better solution, if available, is negative lookahead. Perl provides this with its ⌈`x(?!...)`⌋ construct (☞ 228): an `x` not followed by a slash would be ⌈`x(?!/)`⌋.

Lookbehind, unfortunately, is not offered in any regex flavor that I know of, so for "slash, so long as not preceded by `x`" you can't do much except to prepend ⌈`(^|[^x])/`⌋.

Phew! Somewhat confusing, isn't it? Real comments (with * instead of `x`) require

> ⌈`/\*([^*]|\*+[^/*])*\*+/`⌋

which appears even worse. It's not easy to read; just remember to keep your wits about you as you carefully parse complex expressions in your mind.

Unrolling the C Loop

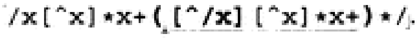
For efficiency's sake, let's look at unrolling the regex. Table 5-3 on the next page shows the expressions we can plug in to our unrolling-the-loop pattern. Like the subdomain example, the ⌈*normal* *⌋ is not actually free to match nothingness. With subdomains, it was because the normal part was not allowed to be empty. In this case, it's due to how we handle the two-character ending delimiter. We ensure that any *normal* sequence ends with the first character of the ending delimiter, allowing *special* to pick up the ball only if the following character does not complete the ending.

Table 5-3: Unrolling-The-Loop Components for C Comments

⌜`opening normal* ( special normal* ) * closing,`⌟

Item	What We Want	Regex
opening	start of comment	`/x`
normal*	comment text up to, and including, one or more 'x'	`[^x]*x+`
special	something other than the ending slash (and not 'x')	`[^/x]`
closing	trailing slash	`/`

So, plugging these in to the general unrolling pattern, we get:



`/x[^x]*x+([^/x][^x]*x+)*/`.

Notice the spot I have marked? The regex engine might work to that spot in two ways (just like the expression on page 167). The first is by progressing through after the regex's leading ⌜`/x[^x]*x+`⌟, or by looping due to the `(…)*`. Via either path, once we're at that spot we know we've matched x and are at a pivotal point, possibly on the brink of the comment's end. If the next character is a slash, we're done. If it's anything else (but an `x`, of course), we know the `x` was a false alarm and we're back to matching normal stuff, again waiting for the next `x`. Once we find it, we're right back on the brink of excitement at the marked spot.

Return to reality

⌜`/x[^x]*x+([^/x][^x]*x+)*/`⌟ is not quite ready to be used. First, of course, comments are `/*…*/` and not `/x…x/`. This is easily fixed by substituting each `x` with `\*` (or, within character classes, each `x` with `*`):

⌜`/\*[^*]*\*+([^/*][^*]*\*+)*/`⌟

A use-related issue is that comments often span across lines. If the text being matched contains the entire multi-line comment, this expression should work. With a strictly line-oriented tool such as *egrep,* though, there is no way to apply a regex to the full comment (anyway, most *egreps* use a DFA engine, so there's no need to bother with this unrolling-the-loop technique to begin with). With Emacs, Perl, and most other utilities mentioned in this book, you can, and this expression might be useful for, say, removing comments.

In practical use, a larger problem arises. This regex understands C comments, but does not understand other important aspects of C syntax. For example, it can falsely match where there is no comment:

```
const char *cstart = "/*", *cend = "*/";
```

We'll develop this example further, right in the next section.

Other Quantifiers: Repetition

Similar to the question mark are ⌜+⌟ (*plus*) and ⌜*⌟ (an asterisk, but as a regular-expression metacharacter, I prefer the term *star*). The metacharacter ⌜+⌟, means "one or more of the immediately-preceding item," and ⌜*⌟ means "any number, including none, of the item." Phrased differently, ⌜...*⌟ means "try to match it as many times as possible, but it's okay to settle for nothing if need be." The construct with plus, ⌜+⌟, is similar in that it will also try to match as many times as possible, but different in that it will fail if it can't match at least once. These three metacharacters, question mark, plus, and star, are called *quantifiers* (because they influence the quantity of a match they govern).

Like ⌜...?⌟, the ⌜...*⌟ part of a regular expression will always succeed, with the only issue being what text (if any) will be matched. Contrast this to ⌜...+⌟ which fails unless the item matches at least once.

An easily understood example of star is ⌜•*⌟, the combination with a space allowing optional spaces. (⌜•?⌟ allows one optional space, while ⌜•*⌟ allows any number.) We can use this to make page 9's `<H[1-6]>` example flexible. The HTML specification* says that spaces are allowed immediately before the closing >, such as with `<H3•>` and `<H4•••>`. Inserting ⌜•*⌟ into our regular expression where we want to allow (but not require) spaces, we get ⌜`<H[1-6]•*>`⌟. This still matches `<H1>`, as no spaces are required, but it also flexibly picks up the other versions.

Exploring further, let's search for a particular HTML tag recognized by Netscape's World Wide Web browser *Navigator*. A tag such as `<HR•SIZE=14>` indicates that a line (a Horizontal Rule) 14 pixels thick should be drawn across the screen. Like the `<H3>` example, optional spaces are allowed before the closing angle bracket. Additionally, they are allowed on either side of the equal sign. Finally, one space is required between the `HR` and `SIZE`, although more are allowed. For this last case, we could just insert ⌈`•• *`⌋, but instead let's use ⌈`•+`⌋. The plus allows extra spaces while still requiring at least one. Do you see how this will match the same as ⌈`••*`⌋? This leaves us with ⌈`<HR•+SIZE•*=•*14•*>`⌋.

Although flexible with respect to spaces, our expression is still inflexible with respect to the size given in the tag. Rather than find tags with only a particular size (such as `14`), we want to find them all. To accomplish this, we replace the ⌈`14`⌋ with an expression to find a general number. Well, a number is one or more digits. A digit is ⌈`[0-9]`⌋, and "one or more" adds a plus, so we end up replacing ⌈`14`⌋ by ⌈`[0-9]+`⌋. As you can see, a single character class is one "unit", so can be subject directly to plus, question mark, and so on, without the need for parentheses.

---

* If you are not familiar with HTML, never fear. I use these as real-world examples, but I provide all the details needed to understand the points being made. Those familiar with parsing HTML tags will likely recognize important considerations I don't address at this point in the book.

The Freeflowing Regex

We just spent some time constructing a regex to match a C comment, but left off with the problem of how to stop comment-like items within strings from being matched. Using Tcl, we might mistakenly try to remove comments with:

```
    set COMMENT  {/\*[^*]*\*+([^/*][^*]*\*+)*/} #   regex to match a
comment
    regsub -all "$COMMENT" $text {} text
```

(*note: Tcl's non-interpolative strings are given as* `{...}`)

This takes the contents of the variable `COMMENT`, interprets it as a regex, and replaces all matches of it in the string given by the variable `text`. Matched text is replaced by nothing (denoted by Tcl's empty string, `{ }`). When done, the results are placed (back) into the variable named `text`.

The problem is that when the regex doesn't match at any particular starting point, the transmission does its normal bump-along, and that could bring the start of the match through to inside a string. It would be nice if we could tell the regex engine that when it hits a doublequoted string, it should zip right on past it.

A Helping Hand to Guide the Match

Consider:

```
    set COMMENT {/\*[^*]*\*+([^/*][^*]*\*+)*/}   #  regex to match a
comment
    set DOUBLE  {"(\\.|[^"\\])*"}                #  regex to match
doublequoted string
    regsub -all "$DOUBLE|$COMMENT" $text {} text
```

When the match start is at a position where the `$DOUBLE` part can match, it will do so, thereby bypassing the whole string in one fell swoop. It's possible to have both alternatives because they're entirely unambiguous with respect to each other. Reading from the left, any starting point in the string. . .

- is matchable by the comment part, thereby skipping immediately to the end of the comment, or. . .

- is matchable by the doublequoted-string part, thereby skipping immediately to the end of the string, or. . .

- is not matchable by either, causing the attempt to fail. This means that the normal bump-along will skip only the one, uninteresting character.

This way, the regex will never, ever, be *started* from *within* a string or comment, the key to its success.

Actually, right now it isn't helpful yet, since it removes the strings as well as the comments, but a slight change puts us back on track.

Consider:

```
    set COMMENT  {/\*[^*]*\*+([^/*][^*]*\*+)*/}      #   regex to
match a comment
    set DOUBLE   {"(\\.|[^"\\])*"}                   #   regex to
match doublequoted string
    regsub -all "($DOUBLE)|$COMMENT" $text {\1} text
```

The only differences are that we:

- Added the parentheses to fill `\1` (which is Tcl's replacement-string version of Perl's `$1`) if the match is the string alternative. If the match is the comment alternative, `\1` will be left empty.

- Made the substitute string that same `\1`. The effect is that if a doublequoted string is matched, the replacement will be that same doublequoted string—it is not removed because the match and substitute becomes an effective no-op. On the other hand, if the comment alternative is the one that matches, the \1 will be empty, so the comment will be replaced by nothingness just as we want.

Finally, we need to take care of singlequoted C constants such as `'\t'` and the like. This is no problem—we simply add another alternative inside the parentheses. If we would like to remove C++ style `//` comments too, that's as simple as adding ⌈`//[^\n]*`⌋ as a fourth alternative, outside the parentheses. Tcl regexes don't support `\n`, but Tcl doublequoted strings do, so we have to use the string `"//\[^\n]*"` to create ⌈`//[^␤]*`⌋:

```
    set COMMENT  {/\*[^*]*\*+([^/*][^*]*\*+)*/}   #   regex to match a
comment
    set COMMENT2 "//\[^\n]*"                      #   regex to match a
C++// comment
    set DOUBLE   {"(\\.|[^"\\])*"}                #   regex to match
doublequoted string
    set SINGLE   {'(\\.|[^'\\])*'}                #   regex to match
singlequoted constant
    regsub -all "($DOUBLE|$SINGLE) |$COMMENT|COMMENT2" $text
{\1} text
```

The basic premise is quite slick: when the engine checks the text, it quickly grabs (and if appropriate, removes) these special constructs. I did a quick test: on my machine, Tcl takes about 37 seconds to remove all the comments from a 60k+ line, 1.6 megabyte test file.

A Well-Guided Regex is a Fast Regex

With just a little hand-holding, we can help direct the flow of the regex engine's attention to match much faster. Let's consider the long spans of normal C code between the comments and strings. For each such character, the regex engine has to try each of the four alternatives to see whether it's something that should be gobbled up, and only if all four fail does it bump-along to bypass the character as uninteresting. This is a lot of work that we really don't need to do.

We know, for example, that for any of the alternatives to have a chance at matching, the lead character must be a slash, a singlequote, or a doublequote. One of these doesn't guarantee a match, but *not* being one does guarantee a non-match.

So rather than letting the engine figure this out the slow and painful way, let's just tell it directly by adding ⌈`[^'"/]`⌋ as an alternative. In fact, any number of these puppies in a row can be scooped right up, so let's use ⌈`[^'"/]+`⌋ instead. If you remember the neverending match, you might feel worried about the added plus. Indeed, it could be of great concern if it were within some kind of `(...)*` loop, but in this standalone case it's quite fine (there's nothing which follows that could force it to backtrack in search of a neverending match). So adding

```
  set OTHER    {[^"'/]}   # Stuff that couldn't possibly begin one of the other
alternatives
    ⋮
  regsub -all "($DOUBLE|$SINGLE|$OTHER+)|$COMMENT|$COMMENT2"
$text {\1} text
```

I retry my benchmarks, and lo and behold, this one change cuts the time from 36 to under 3.2 seconds! That's an order of magnitude right there. We've crafted the regex to remove most of the overhead of having to try all the alternatives and then bumping along. There are still a few cases where none of the alternatives will match (such as at '`c / 3.14`'), and at such times we'll have to be content with the bump-along to get us by.

However, we're not done yet-we can still help the engine flow to a faster match:

- In most cases, the most popular alternative will be ⌈`$OTHER+`⌋, so let's put that first inside the parentheses. This isn't an issue for a POSIX NFA engine because it must always check all alternatives, but for Tcl's Traditional NFA which stops once a match has been found, why make it hunt for what we believe will be the most popular alternative?

- After one of the quoted items matches, it will likely be followed by some `$OTHER` before another string or a comment is found. If we add ⌈`$OTHER*`⌋ after each string, we tell the engine that it can immediately flow right into matching `$OTHER` without bothering with the `-all` looping. This is similar to the unrolling-the-loop technique. In fact, unrolling the loop gains much of its speed from the way it leads the regex engine to a match, using our global knowledge to create the local optimizations that feed the engine just what it needs to work quickly. It's all part of the same cloth.

Note that it is *very* important that this `$OTHER`, added after each string-matching subexpression, be quantified with star, while the previous `$OTHER` (the one we moved to the head of the alternation) be quantified by plus. If it's not clear, consider what could happen if the appended `$OTHER` had plus and there were two, say, doublequoted strings right in a row. Also, if the leading used star, it would always match!

These changes yield

```
"($OTHER+|$DOUBLE$OTHER*|$SINGLE$OTHER*)|$COMMENT|$COMMENT2"
```

as `regsub`'s regex argument, and cut the time another six percent or so, to about 2.9 seconds. So far, we've sped things up twelvefold just by helping the match flow.

Let's step back and think about these last two changes. If we go to the trouble of scooping up `$OTHER*` after each quoted string, there are only two situations the original `$OTHER+` (which we moved to be the first alternative) can match in: *1)* at the very start of the whole `regsub` before any of the quoted strings get a chance to match, and *2)* after any comment. You might be tempted to think "Hey, to take care of point #2, let's just add `$OTHER*` after the comments as well!" This would be nice, except everything we want to keep must be inside that first set of parentheses—putting it after the comments would throw out the baby code with the comment bathwater.

So, if the original `$OTHER+` is useful primarily only after a comment, do we really want to put it first? I guess that depends on the data—if there are more comments than quoted strings, then yes, placing it first makes sense. Otherwise, I'd place it later. As it turns out with my test data, placing it first yields better results. Placing it later takes away about half the gains we achieved in the last step.

Wrapup

Are we done yet? Not on your life! Don't forget, each of the quoted-string subexpressions is ripe for unrolling—heck, we spent a long section of this chapter on that very topic. Replacing the two string subexpressions by:

```
set DOUBLE      {"[^"\\]*(\\.[^"\\]*)*"}
set SINGLE      {'[^'\\]*(\\.[^'\\]*)*'}
```

yields results around 2.3 seconds, yet another 25 percent gain. Figure 5-9 shows all our gains thus far.

Using variables to build up regular expressions in Tcl is a pleasure because the `{…}` non-interpolative quoting construct is so consistent. That's a good thing, too, since the raw expression is ridiculously long, as shown here broken across lines to fit the page:

```
([^"'/]+|"[^"\\]*(\\.[^"\\]*)*"[^"'/]*|'[^'\\]*(\\.[^'\\]*)*'["'/]*)|
    /\*[^*]*\*+([^/*][^*]*\*+)*/|//[^\n]*
```

Perl's analogous singlequoted string, for example, is not as pleasant or consistent in that `\\` means `\` and not `\\`. Perl offers other ways to generate readable regexes, as we'll see when we revisit this example with Perl in Chapter 7 (☞ 292).

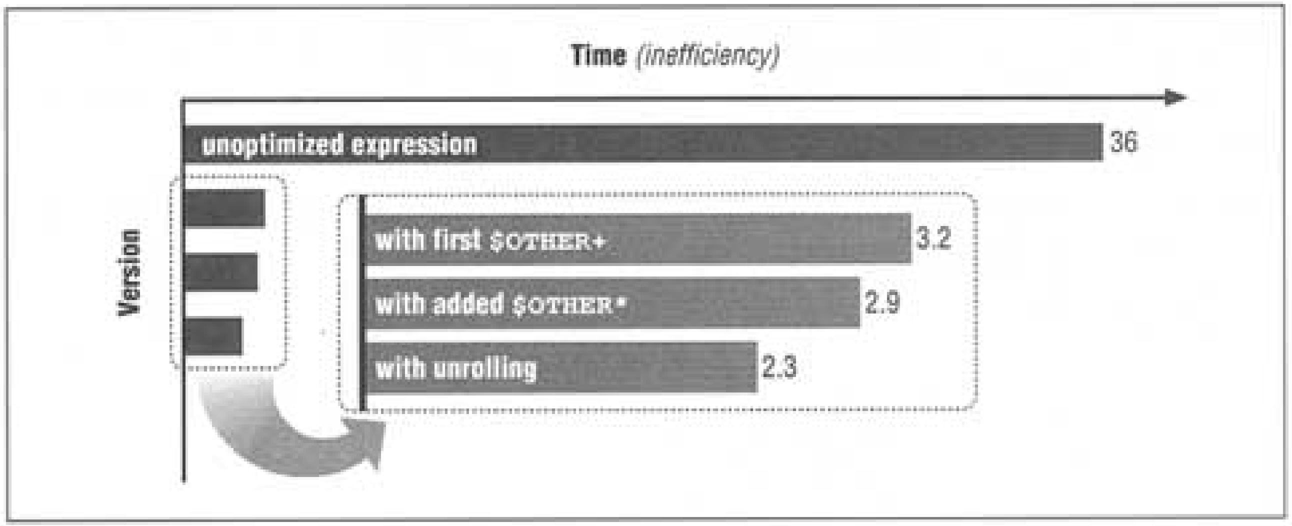


Figure 5-9:
Well-guided optimization gains

Think!

More important than general optimizations, an NFA engine's best efficiency is probably found in a regex crafted with the knowledge of just what you do and don't need to achieve your goal, tempered with the knowledge of what a NFA engine needs to do to get it done.

Let's look at an example I ran into recently while using Emacs, which has a Traditional NFA engine. I wanted a regex to find certain kinds of contractions such as "don't," "I'm," "we'll," and so on, but not other situations where a single quote might be next to a word. I came up with a regex to match a word, ⌜`\<\w+`⌟, followed by the Emacs equivalent of ⌜`'([tdm]|re|ll|ve)`⌟. It worked, but I realized that using ⌜`\<\w+`⌟ was silly when I only needed `\w`. You see, if there is a `\w` immediately before the apostrophe, `\w+` is certainly there too, so having the regex check for something we know is there doesn't add any new information unless I want the exact extent of the match (which I didn't, I merely wanted to get to the area). Using `\w` alone made the regex more than 10 times faster.

The Many Twists and Turns of Optimizations

Consider using

⌈[\b**char**\b|\b**const**\b|\b**double**\b...\b**signed**\b|\b**unsigned**\b|\b**while**\b⌋

to find lines with certain C reserved words in them. From a purely regex point of view, the alternation is costly—each alternative is attempted at each position in the string until a match is found.

On the other hand, if the regex were only one word, such as with ⌈`\bchar\b`⌋, the common *fixed-string-check* optimization (☞ 155) can be employed to allow the engine to quickly scan for the one match. The sum of a series of such checks is often faster than the one large check.

As a specific example, I created a short Perl script to count lines in the Perl source distribution that had these words. One version looks like:

```
$count = 0;
while (<>)
{
   $count++ if m<  \bchar\b
                 | \bconst\b
                 | \bdouble\b
                    ⋮
                 | \bsigned\b
                 | \bunsigned\b
                 | \bwhile\b      >x;
 }
```

This uses the `/x` feature of Perl, which allows free-form spacing within non-class parts of the regex. Everything but the whitespace within the long, multi-line ⌈`m<…>x`⌋ is one regex (with many alternatives). Another test looks like:*

```
$count = 0;
while (<>) {
   $count++, next if m/\bchar\b/;
   $count++, next if m/\bconst\b/;
   $count++, next if m/\bdouble\b/;
     ⋮
   $count++, next if m/\bsigned\b/;
   $count++, next if m/\bunsigned\b/;
   $count++, next if m/\bwhile\b/;
}
```

Both versions count the same number of lines, but the second does so over six times faster. It eliminates much of the backtracking overhead, and as a bonus makes each individual match amenable to the internal optimizations.

Things are not always so clear cut. As I mentioned on page 154, GNU Emacs is very good at the *first-character-discrimination* optimization—much, much better than Perl, Tcl, Python, or any other NFA-based tool that I know of. With the many-alternatives case, the transmission knows to apply the full regex only at places in

---

* Perl note The use of

```
    statement, next if conditional;
```

is a common and useful idiom meaning:

```
    if (conditional) {
       statement;
       next        #restart enclosing loop, like C's 'continue'
    }
```

the string where ⌈`[cdsuw]`⌋ could match, as only those characters begin the alternatives (well, with the full data of the test, not shown, `[cdfinrsuw]` is used). It can be a huge win to not have to put the full engine into motion at each position. Were the regex to start with ⌈`.*`⌋, or some other construct allowing a match to start with any random character, the optimization would be lost, but it's applicable in this example: the same set of tests has Emacs running the many-alternatives version 3.8 times faster than the many-regex version.

Ah, but things *still* aren't so clear-cut. As I mentioned in "Compile Caching" (☞ 158), Emacs caches only the five most recently used regexes. In the manyregex version, using a separate regex to check each word (there are 14 in my test) makes the cache virtually useless. Increasing the size to be large enough to cache all the regexes results in a threefold speedup! This brings the relative time of the many-regex test from 3.8 times slower than the many-alternatives version down to just 1.4 times slower, which more realistically illustrates Emacs's *first-character discrimination.*

By the way, note that the comparison between the many-alternative versions and the many-regex version is possible because we care only *whether* there is a match, not *where* the match is. A single expression like ⌈`char|const|…`⌋ finds the first word on the line (without regard to the ordering of the list), while separate checks find the first word in the list (without regard to the location in the line). It's a huge difference, but one we know we don't care about in this example.

Targeting other optimizations

Continuing along the same lines, but manually, we can try to bring the common leading element of each alternative out to the front of the alternation. This is similar to changing ⌈`this|that|the•other`⌋ to ⌈`th(is|at|e•other)`⌋. Such a change allows the relatively slow alternation to get a chance only after ⌈`th`⌋ matches, even if *first-character discrimination* is not used. It also creates an easy-to-detect common fixed string, '`th`', for a *fixed-string-check* optimization. That's a big savings. With the reserved words example, this entails bringing the ⌈`\b`⌋ from the front of each alternative to the head of the regex: ⌈`\b(char\b|const\b|…)`⌋. With my Perl benchmarks, the result after this simple change is almost as fast as the separatecheck method given above.

The same idea can be used outside of regular expressions, using other operators the host language might provide. For example:

```
/this/ || /that/ || /the other/
```

The `||` outside the regexes means "or". The regexes will be tested in turn until one matches, at which point the whole …`||`…`||`… expression stops. This leaves intact all the side effects of the first match to succeed.

When it comes down to it, thought and logic take you most, but not necessarily all, the way to an efficient program. Without benchmarking the Perl C reserved words example, I could easily tell you that either of the faster methods would be faster than the slower one, but I would be hard pressed to say which of the faster is fastest. Perhaps another internal optimization in the next version of Perl, or whatever tool I happen to test these with, will change which is fastest.

6

Tool-Specific Information

In this chapter:

- ***Questions You Should Be Asking***
- ***Awk***
- ***Tcl***
- **GNU Emacs**

Knowing regular expressions in general is only half of the story—the specific flavor, limitations, and potential of the host utility complete it.

As Chapter 3 begins to show, there's quite a bit of variety out there. Table 3-4 (☞ 82) illustrates that even simple things like line anchors can be complicated. To give a further sample, Table 6-1 on the next page shows a few regex flavor characteristics across a collection of tools available for a wide variety of platforms. You might have seen a watered-down version of a chart like this before (such as, say, this very book's Table 3-1), but don't let the size of Table 6-1 fool you—it's also watered down. There should be footnotes clarifying almost every entry.

Questions You Should Be Asking

When you begin to inspect an unknown regex flavor, many questions should come to mind. A few are listed in Chapter 3 (☞ 63). Although these issues aren't all hot, front-burner concerns, you should be familiar with them. One obstacle is that different versions of the same tool can vary considerably. For example, GNU versions of software tend to brim with extra features. Table 6-1 does not reflect this, nor the whole breed of POSIX-compliant versions. In fact, although your version of one of the generic tools listed in Table 6-1 might match the chart exactly, chances are slim. Even something as simple as *grep* varies widely.

Something as Simple as Grep. . .

Yes, even something as simple as *grep* varies widely. Table 6-2 on page 183 looks at some of the differences among just a few of the many versions out there. Table 6-2, too, should have its fair share of footnotes. With GNU *grep* version 2.0, `^` and `$` are line anchors wherever they "make sense," such as in ⌜`…|^…`⌟ and ⌜`(…$)`⌟

Table 6-1. A Superficial Survey of a Few Common Programs' Flavor

	Trad. *grep*	*vi*	Modern *grep*	Modern *sed*	*egrep*	*lex*	Trad. *awk*	GNU Emacs	Perl	Tcl	(default) Python	Expect
engine type	NFA	NFA	DFA	NFA	DFA	DFA	DFA	NFA	NFA	NFA	NFA	NFA
dot matches 'NL'	N/A	N/A	N/A	√	N/A	•	√	•	•	√	•	√
octal hex	•	•	•	•	•	√ · •	√	✐	√	✐	✐	✐
`[…],^,$`	√	√	√	√	√	√	√	√	√	√	√	√
`\<…\>` `\b,\B`	•	•	•	√ · •	√ · √	•	•	√ · √	€ • ·√	√	√€ · √	•
`grouping`	`\(…\)`	`\(…\)`	`\(…\)`	`\(…\)`	`(…)`	`(…)`	`(…)`	`\(…\)`	`(…)`	`(…)`	`\(…\)`	`(…)`
`(…)*`	•	•	√	√	√	√	√	√	√	√	√	√
`+` or `\+`	•	•	\+	\+	+	+	+	+	+	+	+	+
`?` or `\?`	•	•	\?	\?	?	?	?	?	?	?	?	?
`{…}` or `\{…\}`	•	•	`\{…\}`	`\{…\}`	`{…}`	`{…}`	•	•	`{…}`	•	•	•
`|` or `\\|`	•	•	`\\|`	`\\|`	`|`	`|`	`|`	`\\|`	`|`	`|`	`\\|`	`|`
`\w, \W`	•	•	•	•	√	•	•	√	√	•	•	•
`\n, \t`	•	•	•	•	•	√	√	✐	√	✐	✐	✐
lookahead	•	•	•	•	•	limited	•	•	√	•	•	•
max. backrefs	9	9	9	9	•	•	•	9	∞	•	99	•
null okay	•	•	•	•	•	•	•	√	√	√	√	√
case insensitive	√	√	√	•	√	•	•	√	√	√	√	√

situations. With most other versions, they are valid only at the start and end of the regex, respectively. On another front, GNU *grep* version 2.0 lacks the POSIX class `[:blank:]`. SCO's lacks `[:xdigit:]`.

The `-i` command-line option seems to be fertile ground for differences. With the 1979 `v7` version of *grep*, `-i` applied only to characters in the regex that were already lower case. The `r2v2` version a few years later fixed that, but then it didn't work for characters matched by a character class. Even GNU's modern *grep* version 2.0 doesn't apply `-i` when matching a backreference. SCO's doesn't when matching the `[:upper:]` POSIX class, but does when matching `[:lower:]`. Why? Your guess is as good as mine.

Well, I think you get the point by now: Differences abound.

In This Chapter

No one could ever hope to keep track of every nuance in every version of every implementation of every program out there. This includes me, which is why Table 6-1 is as far as I'll go along those lines for most of the tools listed there. When it comes down to it, once you internalize Chapters 4 and 5, you become aware of the most important concerns, and skilled enough to investigate and make sense of them yourself. Other than these questions, there's usually not much to talk about when using any one specific tool, other than what you get in the manual. Perl is a notable exception, so I cover it in the next chapter.

Table 6-2: A Comical Look at a Few Greps

	UNIX **v7**	UNIX **r2v2**	GNU grep **1.2**	GNU grep **2.0**	SCO **sVr3**	DG/UX	MKS **(for NT)**
`\(...\)` (flat/nested)	9/9	9/0	9/9	∞/∞	9/0	9/9	∞/∞
`(...) *` allowed	•	•	•	√	•	•	√
`^,$`	√	√	√	√	√	√	√
`\+,\?`	•	•	•	√	•	•	√
`|,`NL	•	•	•	√	• · √	•	• · √
`\<...\>, \b`	•	•	√ · •	√	•	√ · •	√ · •
`\{`*min, max*`\}`	•	√	•	√	√	√	√
`\w,\W`	•	•	•	√	•	•	•
POSIX equiv, collate, class	•	•	•	• · • · √	*all*	• · √ · √	*all*
'NL'-friendly, NULLs	√ · √	√ · •	• · •	√ · √	√ · •	√ · •	√ · •
`-i`	`-i`	`-i,-y`	`-i`	`-i,-y`	`-i,-y`	`-i,-y`	`-i`
NFA/DFA	NFA	NFA	NFA	BOTH	NFA	NFA	NFA

As a sampling, this chapter looks at a few of the regex—related specifics of *awk*, Tcl, and GNU Emacs's *elisp.* Each section assumes that you are reasonably familiar with the target language, and that you are looking for either a concise description of the regex flavor or maybe just a few hints and tips. I've tried to keep them short and to the point, so I often refer you to Chapter 3.

The first section, *awk,* focuses on differences among the regex—related features of some popular implementations. *awk* is a generic utility put out by many vendors, and they each seem to have their own idea about just what the regex flavor should be. Discussions of portability are lively—it can be either shocking or amusing, depending on how you look at it.

On the other hand, Tcl and GNU Emacs each comes from just one source, so your copy is probably the same, or at least largely similar, as everyone else's. You might need to be concerned about differences among versions, but in general, it's more interesting to talk about their approach to regular expressions, their NFA engine, and their efficiency.

Awk

awk, created during a short week of hacking in 1977, was the first Unix power tool for text manipulation. Much more general and expressive than *sed,* it created a whole new culture that carries on into the next generation of tools. The original authors combined various interests into one tool: Alfred Aho, who had just written *egrep* and had a strong hand in *lex,* brought regular expressions. Peter Weinberger had an interest in databases, and Brian Kernighan had an interest in programmable editors. A tool by Marc Rochkind was a strong influence. It converted *regex-string* pairs into a C program to scan files and print the corresponding *string* when a *regex* matched. *awk* greatly expanded on this idea, but the central roles of matching input line-by-line and of regular expressions are the same.

awk's regex flavor resembled that of *egrep's* more than any other tool available at the time, but it was *not* the same. Unfortunately, that didn't stop the manpage from saying it was, a myth that continues today (even in some O'Reilly books!). The differences can cause confusion. Some differences are readily apparent—*awk* supports ⌜`\t`⌟ and ⌜`\n`⌟, *egrep* doesn't (although *awk's* support for these wasn't even mentioned in the original documentation!). There are other, less obvious differences as well. Saying the two flavors were the same hid important features of *awk* and confused users that ran into them.

Differences Among Awk Regex Flavors

Twenty years ago, Bell Labs' version of *awk* was the only one. Today, there are a plethora. Over the next few pages, I'll take a detailed look at some of the differences among a few selected implementations. The idea is not to relate specific information about the implementations and differences I happen to show, but to tangibly illustrate that beauty is more than skin deep, to borrow a phrase. There are many other implementations (and differences) in addition to these, and many of these might not persist to future versions.

In particular, as this book was nearing publication, the maintainer of GNU *awk*, Arnold Robbins, was going through an earlier manuscript, fixing bugs I reported about the version `3.0.0` used while preparing this chapter. Accordingly, some bugs have been fixed in later versions (comments on a few of which I've been able to include in this printing).

Table 6-3: A Superficial Look at a Few Awks

	oawk	**nawk**	**awk**	**gawk**	**MKS**	**mawk**
engine type	DFA	DFA	DFA	both	NFA	NFA
true longest-leftmost matching	√	√	√	√	√	√
basic`(...) . | * + ?[...]`	√	√	√	√	√	√
`\<` `\>` word boundaries	•	•	•	√	√	•
escaped newline in regex	⌜|⌟	fatal	fatal	⌜|⌟	⌜|⌟	⌜|⌟
`\b, \f, \n, \r, \t` · `\a, \v`	`\t, \n`	√ · •	√ · •	√	√	√
`\w, \W` · ⌜`{`*`min, max`*`}`⌟ · backreferences	•	•	•	√ · √ · •	• · • · √	• ·€ √ · •
can use escapes in class	only `\]`	all but `\-`	√	√	none	√
`\123` octal, `\xFF` hex escapes	•	√ · •	√	√	√	√
dot matches newline, anchors ignore them	√	√	√	√	√	√
POSIX `[:...:]`, etc.	•	•	•	`[:...:]` only	√	•
can do case-insensitive matches	•	•	•	√	•	•

Some of the more visible differences among the *awks* are shown in Table 6-3. It compares the original *awk,* and several popular versions available today:

- **oawk**—the original *awk,* as distributed by AT&T with Unix Version 7, dated May 16, 1979.
- **nawk**—*new awk,* as distributed with SCO Unix Sys V `3.2v4.2`.
- **awk**—*the One True Awk,* still maintained and distributed by Brian Kernighan. Tested version: `June 29, 1996.`
- **gawk** —GNU *awk.* Tested version: `3.0.0`.
- **MKS** Mortice Kern Systems' *awk* for Windows-NT.
- **mawk**—*Mike's awk,* by Michael Brennan. Tested version: `1.3b`.

A bold name refers to the specific program mentioned in the list. Thus, *awk* refers to the generic utility, while awk refers to Brian Kernighan's *One True Awk.*

Again, don't let Table 6-3 fool you into thinking you have the complete information you need to write portable *awk* scripts. There are *many* other concerns. Some are quite minor. For instance, only in **oawk, nawk,** and **awk,** must you escape an equal sign that starts a regex (seems odd, doesn't it?). Some differences are quite tool-specific (for example, you can use the `{`*`min,max`*`}` quantifier with **gawk** only if you use the `--posix` or `--re-interval` command-line options). However, some major, if not sometimes subtle differences tend to involve most implementations. The next few subsections examine a few of them.

Awk octal and hexadecimal escapes are literal?

With versions of *awk* that support hexadecimal escapes, you might reasonably expect ⌜`ora\x2Ecom`⌟ to match the same as ⌜`ora\.com`⌟. (`2E` is the ASCII code for a period.) As discussed in Chapter 3 (☞ 75), the reasoning is that if you go to the trouble to use a hexadecimal or octal escape, you wouldn't want its result to be taken as a metacharacter. Indeed, this is how **awk** and **mawk** work, but **gawk** and **MKS**, as per POSIX, actually treat ⌜`\x2E`⌟ as a dot metacharacter (although **gawk** does not if you use the `--traditional` command-line option).

Do octal and hexadecimal escapes show restraint?

You might also expect the hexadecimal escape in ⌜`ora\x2Ecom`⌟ would be `\x2E`, but some implementations suck up all hexadecimal digits after `\x`, so in this case the escape would actually be `\x2ec` (more details ☞ 74). Again, both **gawk** and **MKS** do this (although **gawk** didn't allow single-digit hexadecimal escapes until version `3.0.1`).

It's worse with octal escapes. All allow one-, two-, and three- digit octal escapes (except **MKS**, which uses one-digit escapes for backreferences—a feature not mentioned in their documentation), but the similarities stop there. **awk**, **MKS**, and **mawk** properly ignore an `8` or `9` after a one- or two- digit octal escape, **nawk** considers `8` and `9` to be octal! (☞ 74). **gawk** handles a trailing `8` and `9` properly, but issues a fatal error for `\8` and `\9` (an example of one of the bugs fixed in a version slated for release).

Empty regex or subexpression in awk

It's certainly reasonable to want something like ⌈`(this|that|)`⌋, where the empty subexpression means to always match (☞ 84). With a DFA, it's exactly the same as ⌈`(this|that)?`⌋, but sometimes for notational convenience or clarity, the empty-subexpression notation could be preferred. Unfortunately, not all implementations allow it: **awk**, **mawk,** and **nawk** consider it a fatal error.

An empty regex altogether is a different thing. Both **awk** and **nawk** still consider it a fatal error, while **mawk, gawk,** and **MKS** have it match any non-empty string.

An awk regex with consistent class

Perhaps the biggest can of worms is the (in)ability to escape items in a character class, and the treatment of'`]`' and '`-`' inside a character class. I covered escapes in Table 6-3,[*] but consider a class that starts with '`]`'. No problem with **awk, gawk,** and **MKS**, but it's a fatal error with **mawk**, and is simply ignored by **nawk** (as it was with **oawk).**

What happens if you mistakenly specify an inverted range like ⌈`[z-a]`⌋? A more realistic example is ⌈`[\-abc]`⌈, where you thought **`\-`** meant a literal dash instead of the beginning of the range from `\` to `a` (which is probably not useful, since a backslash comes *after* `a` in the ASCII encoding--an inverted range). **gawk** and **MKS** consider such ranges to be a fatal error; **awk** inverts the range automatically; **mawk** treat the whole thing as a non-range (in other words, as two literal characters: a dash and an `a`). If that's not enough variation, **nawk** doesn't complain, but only includes the starting character in the class (in this example, the backslash).

What kind of data can awk process

Some implementations have limitations as to what kind of data can be processed. **nawk** and **awk** don't allow anything other than 7-bit ASCII (in other words, bytes with the high bit set can never match), while only **gawk** allows nulls to be processed. **(MKS** considers a null byte to be a fatal error, while the others just treat it as the end of the line or regex, as the case may be.)

---

* Well, one thing not covered in Table 6-3 is the escaping of slash, the regex delimiter, within a class. With the original **awk**, you could if you wished, but didn't need to. With **gawk**, you may not, and with the others, you must.

Awk Regex Functions and Operators

The modern *awks* (among those under discussion, all but **oawk**) provide at least three ways to use regular expressions: the input-field separator, the ~ and `!~` operators, and the functions `match`, `sub`, `gsub`, and `split` (**gawk** also adds `gensub`). These are simple and well covered in any *awk* documentation, so I won't rehash them here. However, the next few sections discuss a few things to pay particular attention to.

Awk operands: `/…/` versus `"…"`

Most implementations allow either strings `("…")` or raw regexes `(/…/)` to be provided where a regular expression is expected. For example, **`string ~ /regex/`** and **`string ~ "regex"`** are basically the same. One important difference is that in the latter case, the text is first interpreted as a doublequoted string, and then as a regex. This means, for example, that the strings `"\t"` and `"\\t"` both end up being a regex to match a single tab. With `"\t"`, the result of the doublequoted-string processing gives ⌜TAB⌟ to the regex engine (a tab is not a metacharacter, so it will simply match itself), while `"\\t"` provides ⌜`\t`⌟ to the regex engine which interprets it as a metasequence to match a tab. Similarly, both the literal regex `/\\t/` and the string `"\\\\t"` match the literal text '`\t`'.

Contrary to all this, one implementation I checked, **MKS**, treats the two versions (`/…/` and `"…"`) exactly the same. Even though the regex is given as a string, it seems to bypass string processing to be interpreted directly by the regex engine.

Awk regexes that can match nothingness

For the most part, all the implementations I checked agree that a regex must actually match some text in the target string for `split` to actually split at any particular point. (One exception, when there is no regex, is mentioned below.) This means, for example, that using ⌜`,*`⌟ is the same as ⌜`,+`⌟ when applied using `split`.

This does not, however, hold for `sub` and `gsub`. With most versions, running

```
string = "awk"
gsub(/(nothing)*/, "_" string)
```

fills string with '`_a_w_k_`'. However, **gawk** fills it with '`_a_w_k`'. Notice that the final underscore is missing? Until version `3.0.1`, **gawk** didn't match this kind of a `gsub` regex at the end of the string unless ⌈`$`⌋ was involved.

Another issue arises with:

```
string = "sed_and_awk"
gsub(/_*/, "_", string)
```

Most implementations return '`_s_e_d_a_n_d_a_w_k_`', but **gawk** (until version `3.0.1`) and **MKS** return '`_s_e_d_ _a_n_d_ _a_w_k_`'. With these implementations, even after an underscore is matched and replaced, the transmission applies the regex again immediately, resulting in the "⌈`_*`⌋ can match nothing" matching before each '`a`'. (Normally, a match of nothingness is not allowed at the spot the previous match ends.)

Matching whitespace with awk's split(…)

Providing the string `"•"` as the third argument to `split` puts it into a "split on whitespace" mode. For most implementations, *whitespace* means any spaces, tabs, and newlines. **gawk**, however, splits only on spaces and tabs. (As of version `3.0.2`, **gawk** does include newline unless the `--posix` option is used.)

What about when using `/•/` as the regex operand does it cause the special whitespace processing to go into effect? It does with **gawk** (before version `3.0.2`), doesn't with **awk** and **mawk** (and **gawk** from version `3.0.2`), and is a fatal error with **nawk**.

Empty regular expression operands in awk

What does an empty regex operand, such as with `sub("",…)` or `sub(//,…)`, mean? (**awk** never allows an empty `//` regex, but sometimes allows an empty `""` one.) With `gsub`, an empty regex is taken as one that can match everywhere (except with **awk**, where an empty regex to `gsub` is a fatal error). `split`, on the other hand, is a bit more varied: **nawk** and **MKS** don't split at all, while the others split at each character.

Tcl

Tcl[*] uses Henry Spencer's NFA regex package just as he released it, and provides a simple, consistent interface to access it. The regex flavor is direct and uncluttered, and the two regex-related functions are useful and not full of surprises. Don't let the manual's "Choosing Among Alternative Matches" section scare you—Tcl's engine is the good ol' gas guzzling Traditional NFA described in Chapter 4.

---

[*] I'm writing this section as of `Tc17 .5pl`. Tcl's official World Wide Web site: `http://www.sunlabs.com/research/tcl`

Tcl Regex Operands

Let me start right off with an item that might surprise you if you didn't read Chapter 3: Tcl regular expressions do not support 「`\n`」 and friends. Yet, you can use them in most regular expressions. What gives?

Tcl words as regular expressions

Regex operands are normal strings (in Tcl nomenclature, *words*) that happen to be interpreted as regular expressions when passed to the `regexp` or `regsub` functions. This is similar to GNU Emacs and Python. The general discussion of this is in Chapter 3's "Strings as Regular Expressions" (☞ 75). One main consequence is that if the regex is not provided as a string, but is, say, read from a configuration file or is part of a CGI query string, you get none of the string processing, and only what Table 6-4 explicitly shows.

Table 6-4: Tcl's NIFA Regex Flavor

	— Metacharacters Valid Outside a Character Class —
`.`	any byte except null (does include newlines)
`(...)`	grouping and capturing (at most 20 pairs)
`*, +, ?`	standard quantifiers (may govern `(...)`)
`|`	alternation
`^, $`	start- and end-of-string (newlines, or lack thereof, irrelevant)
`\char`	literal char
`[...], [^...]`	normal, negated character class
	Metacharacters Valid Within a Character Class —
`]`	end of class (put first after the `[^` or `[` to include literal '`]`' in class)
`c1-c2`	class range (put first or last to include a literal minus sign in class)

note: within a class, a backslash is completely unspecial

How Tcl parses a script is one of the most basic characteristics of the language one which every user should learn intimately. The `Tcl` manpage gives rather explicit details, so I'll not repeat them here. Of particular interest, however, is the *backslash substitution* processing done on strings not delimited by `{…}`. Backslash substitution recognizes many typical conveniences (Table 3-3, ☞ 73),* and also replaces any escaped newlines (and subsequent spaces and tabs) with single spaces. This happens early in the script processing. Other backslashes are taken as either an escaped delimiter (depending on the type of string), or an unrecognized escaped character, in which case the backslash is simply removed.

---

* Despite the current (as of this writing, `tc17.5`) documentation to the contrary, `\n` in Tcl strings does not necessarily impart a character with hexadecimal value OA. It imparts a system-specific value (☞ 72). However, as this book was going to press, John Ousterhout told me that he intends to hardcode the OA into the program. This could cause future surprises for MacOS users.

The string octal and hexadecimal escapes have a few twists. For example, Tcl considers `8` and `9` to be octal (☞ 74). Tcl hexadecimal escapes allow any number of digits, and also, perhaps unintentionally, support the special sequence `\x0x`*`ddd`*…, which could be surprising, but probably not too surprising, since `\x0` (to mean a null character) is not likely to be used in a Tcl script—Tcl is not generally graceful with nulls in strings.

Using Tcl Regular Expressions

Tcl offers two functions for applying regular expressions, `regexp` for matching, and `regsub` to perform a search-and-replace on a copy of a string. I don't have much to say about them that the manual doesn't already say, so I'll be brief.

Tcl's regexp function

The usage of `regexp` is:

```
    regexp [ options ] regex target [ destination name… ]
```

If the given regex can match within the target string, the function returns `1`, otherwise it returns `0`. If a *destination name* is given, a copy of the text actually matched is placed in the variable with that name. If further names are given, they are filled with the text matched by their respective parenthetical subexpression (or the empty string if no corresponding pair exists, or wasn't part of the match). The named variables are left untouched if the regex doesn't match.

Here's an example:

```
  if [regexp -nocase {^(this|that|other)="([^"]*)"} $string {}
key value] {
     ⋮
```

Here, the `-nocase` is an option that ignores differences in capitalization during the match. The regex ⌜`^(this|that|other)="([^"]*)"`⌟ is checked against the text in `$string`. The `{}` after the target text is an empty place holder where there's normally a name identifying the variable that should receive the entire text matched. I don't need it in this example, so I use the place holder. The next two names, however, receive the text matched by the two parenthetical subexpressions, the `$1` and `$2` of the Perl nomenclature we've seen throughout this book. If `$string` held **`That="l23"•#• sample`**, the variable `$key` would receive `That`, while `$value` would receive `123`. (Had I used a variable name instead of the `{}` place-holder, the variable would have received the entire **`That="123"`** matched.)

Tcl's regsub function

The usage of regsub is:

```
    regsub [ options ] regex target replacement destination
```

A copy of the *target* is placed into the variable named by the *destination* with the first match (or, if the `-all` option used, each match) of the *regex* replaced by the *replacement.* The number of replacements done is returned. If none are done, the destination variable is not modified.

Within the replacement, `&` and `\0` refer to the whole text matched, while `\1` through `\9` each refer to the text matched by the corresponding parenthetical subexpression. Keep in mind, though, `regsub` must see these escape sequences, so extra backslashes or `{…}` are generally required to get them past the main Tcl interpreter.

If you just want to count matches, you can use `{}` as the replacement and destination. The use of a regex that matches anywhere (such as an empty regex, given by `{}`) does the replacement in front of each character. Two ways to underline a string with underline-backspace sequences are:

```
    regsub -all {} $string      _\b      underlined
    regsub -all  . $string      _\b&     underlined
```

Regexp and regsub options

So far, we've seen two options, `-nocase` and `-all`, but there are others. The `regexp` function allows `-indices`, `--`, and `-nocase`, while `regsub` allows `-all`, `--`, and `-nocase`.

The `-indices` option indicates that rather than placing a copy of the *text* matched into the given named variable(s), a string composed of two numbers should be placed instead: zero-based indices into the string where the matched portion begins and ends (or '`-1 -1`' for a non-match). In the `this|that|other` example, `$key` would receive '`0 3`' and $value '`6 8`'.

Despite the documentation to the contrary, `-nocase` works as you might expect, with no surprises. The documentation is worded as if ⌈`US`⌋ wouldn't match '`us`', even if `-nocase` is used.

Normally, all leading arguments that begin with a dash are taken as options (and if unrecognized, as an `error`). This includes situations where the regex is provided from a variable whose contents begins with a dash. Remember, a Tcl function sees its arguments only after all interpolation and such has been done. The special option `--` indicates the end of the options and that the next argument is the regex.

Tcl Regex Optimizations

Tcl places the onus of efficient matching firmly on the programmer's shoulders, as it makes little attempt to optimize regex use. Each regex is re-compiled each time it is seen, although it does cache the compiled versions of the five most recently used regexes. (☞ 159).

The engine itself is just as Henry Spencer wrote it in 1986. Among the optimizations mentioned in Chapter 5 (☞ 154), Tcl tries to do the *first-character discrimination*, a *fixed-string check* (but only if the regex begins with something governed by star or question mark), *simple repetition*, and the recognition that a leading caret can match only at the start of the string. One notable optimization it *doesn't* do is add an *implicit line anchor* when the regex begins with 「.*」.

GNU Emacs

Regular expressions are geared for text processing, so they naturally take a prominent role in one of the most powerful text manipulation environments available today, GNU Emacs* (hereafter, just "Emacs"). Rather than being an editor with a scripting language attached, Emacs is a full programming environment, *elisp*, with a display system attached. The user of the editor can directly execute many *elisp* functions, so these functions can also be considered commands.

Emacs *elisp* (conversationally, just "Lisp") comprises almost a thousand built-in, primitive functions (that is, implemented in C and compiled with the main Emacs system), with standard Lisp libraries providing another dozen thousand implementing everything from simple editing commands like "move cursor to the left" to packages such as news readers, mail agents, and Web browsers.

Emacs has long used a Traditional NFA regex engine, but since Version 19.29 (June 1995), it also supplies longest-leftmost POSIX-like matching as well. Of the thousand built-in functions, four pairs are for applying regular expressions. Table 6-5 lists them and the other regex-related primitives. The popular search features, such as the incremental searches `isearch-forward` and `isearch-forward-regexp`, are Lisp functions that eventually boil down to use the primitives. They might, however, be "value-added" functions—for example, `isearch-forward` converts a single space in the typed regex to ⌈**`\s-+`**⌋, which uses an Emacs *syntax class*, discussed momentarily, to match *any* whitespace.

To keep this section short and to the point, I won't go over the higher-level lisp functions or the specifics of each primitive; this information is as close as the **`C-h f`** keystroke (the `describe-function` command).

---

* This section written as of GNU Emacs Version 19.33.

Emacs Strings as Regular Expressions

Like Tcl and Python, Emacs regular expressions are normal strings that the regex functions pass to the regex engine, which then decides to interpret the contents as a regular expression. The important effects of this are covered in Chapter 3 (☞ 75). Table 6-6 on the next page lists some features of Emacs strings.

Emacs's Regex Flavor

Table 6-7 lists the metacharacters that Emacs's regex engine recognizes. It's an odd mixture of escaped and unadorned metacharacters. Combined with the need for regex escapes themselves to be escaped within a string, the results can often appear to be rows of scattered toothpicks, such as `'"\\(\\\[\\||\\|\\\]\\)"'` (an example taken from real code in the standard Lisp library). See page 76 for more examples.

Table 6-5: GNU Emacs's Search-Related Primitives

Functions to Apply Regular Expressions —

Function	**Purpose**
`looking-at`	applies Traditional NFA regex to current location in buffer
`posix-looking-at`	applies pseudo-POSIX NFA regex to current location in buffer
`string-match`	applies Traditional NFA regex to a given string
`posix-string-match`	applies pseudo-POSIX NFA regex to a given string
`re-search-forward`	searches forward in buffer for a Traditional NFA match
`posix-search-forward`	searches forward in buffer for a pseudo-POSIX NFA match
`re-search-backward`	searches backward in buffer for a Traditional NFA match
`posix-search-backward`	searches backward in buffer for a pseudo-POSIX NFA match
	Literal-String Search Functions —
`search-forward`	searches forward in buffer for a fixed string
`search-backward`	searches backward in buffer for a fixed string
	Search-Result Functions — (available after any of the above are successful)
`match-beginning`	returns position that the match (or subexpression) started
`match-end`	returns position that the match (or subexpression) ended
`match-data`	returns all the positional information from the last match
`store-match-data`	resets the last-match information to the data supplied
`match-string`	returns the text matched by the last regex (or subexpression)
`replace-match`	replaces the *buffer text* last matched with string or pattern

Related function: `regexp-quote`—returns "regex-safe" version of a string.

Related variable: `case-fold-search`—if true, all searches should ignore capitalization differences.

Table 6-6: GNU Emacs's String Metacharacters

\a	ASCII bell	\n	system newline
\b	ASCII backspace	\r	ASCII carriage return
\d	ASCII delete	\t	ASCII tab
\e	ASCII escape	\v	ASCII vertical tab
\f	ASCII formfeed	\A-*char*	Emacs alt-*char*
\C-char	Emacs control-*char*	\H-*char*	Emacs hyper-*char*
\^ char	Emacs control-*char*	\M-*char*	Emacs meta-*char*
\S-char	Emacs shift-*char*	\octal	one- to three-digit octal valued byte
\s-char	Emacs super-*char*	\x*hex*	zero+-digit hexadecimal valued byte
Other \char, including \\\\, places *char* into the string			

Table 6- 7: Emacs's NFA Regex Flavor

	Metacharacters Valid Outside a Character Class —
.	any byte except newline
\\(...\\)	grouping and capturing
*, +, ?	standard quantifiers (may govern \\(...\\))
\\|	alternation
^	start of line (if at start of regex, or after \\| or \\() *matches at start of target text, or after any newline*
$	end of *line* (if at end of regex, before \\|, and next to \\)) *matches at end of target text, or before any newline*
\w, \w	word (non-word) character (see Table 6-8)
\\<, \\>, \b	start-of word, end-of-word, either (see Table 6-8)
\s*code*, \S*code*	character in (not in) Emacs syntax class (see Table 6-8)
\digit	backreference (single-digit backreferences only)
any other \char	literal *char*
[...], [^...]	normal, negated character class
	Metacharacters Valid Within a Character Class —
]	end of class (put first after the [^ or [to include literal ']' in class)
cl-c2	class range (put first or last to include a literal minus sign in class)

note: a backslash is not an escape within a class—it is completely unspecial

Words and Emacs syntax classes

An integral part of Emacs's regex flavor is the *syntax* specification. An Emacs syntax allows the user, or an *elisp* script, to indicate which characters are to be considered a comment, whitespace, a word-constituent character, and so on. The list of *syntax classes* is given in Table 6-8—you can see the details of your current syntax with the command `describe-syntax`, bound, by default, to **`C-h s`**.

A dynamic syntax specification allows Emacs to respond intelligently, under their direction, to the different major modes (`text-mode`, `cperl-mode`, `c-mode`, and the like). The mode for editing C++ programs, for example, defines `/*`…`*/` and `//`…NL as comments, while the mode for editing *elisp* scripts considers only `;`…NL as comments.

The syntax influences the regex flavor in a number of ways. The *syntax class* metacharacters, which combines ⌈`\s…`⌋ and ⌈`\S…`⌋ with the **code** from Table 6-8, provides direct access to the syntax. For example, ⌈`\sw`⌋ is a "word constituent" character; the actual meaning depends on the current mode. All modes consider alphanumerics to be word-constituent characters, but in addition to that, the default `text-mode` includes a singlequote, while `cp ,erl-mode`, for example, includes an underscore.

The syntax for a word-constituent character extends to `\w` and `\W` (simple shorthands for `\sw` and `\Sw`), as well as to the word boundaries, `\<` and `\>`.

Table 6-8: Emacs Syntax Classes

Name	**Code(s)**	**Matches**
charquote	`/`	a character that quotes the following character
close	`)`	an ending-delimiter character
comment	`<`	a comment-starting character
endcomment	`>`	a comment-ending character
escape	`\`	a character that begins a C-style escape
math	`$`	for delimiters like $ in Tex
open	`(`	a beginning-delimiter character
punct	`.`	a punctuation character
quote	`'`	a prefix quoting character (like Lisp `'`)
string	`"`	a string-grouping character (like "…")
symbol	`_`	a non-word-constituent symbol character
whitespace	`-` or ▪	a whitespace character
word	`w` or `W`	a word constituent character

Emacs pseudo-POSIX matching

As mentioned earlier, and as listed in Table 6-5, Emacs provides what it calls POSIX versions of its matching primitives. This does not influence which kind of regular expressions are recognized (that is, the regex flavor does not suddenly become that of Table 3-2; ☞ 64), nor does it influence whether the regex will match or fail. The influence is merely on *which text* will match, and on how quickly it will happen.

On an overall-match level, the POSIX-like versions indeed find the longest of the leftmost matches, just as a true POSIX engine would. Parenthesized subexpressions, however, are *not* maximally filled from the left as described in Chapter 4 (☞ 117). Emacs's NFA engine searches for a match as a normal Traditional NFA, but continues to try all possibilities even when a match is found. It seems that the parentheses state remembered is that of the first match that matched what ended up being the longest match.

For example, during a POSIX-like matching of ⌈`\(12\|1\|123\).*`⌋ against '`1234`', the parenthetically saved text should be `123`, since that is the longest possible submatch within the framework of the longest possible overall match. With an Emacs `posix-` match, however, it's `12`, as that is the first subexpression match that led to a longest match.

Due to the extra overhead of POSIX NFA matches, I strongly recommend against using them unless you specifically need them.

Emacs Match Results

Each function listed at the top of Table 6-5 fills `match-data`, which in turn affects what `match-beginning` and `match-end` return, as well as what `match-string` and `replace-match` work with.

After-match data

The functions **`(match-beginning`** ***num*****`)`** and **`(match-end`** ***num*****`)`** return the positions in the target text where a successful match or subexpression began and ended. If *`num`* is zero, those positions are of the entire match. Otherwise, the positions are for the associated `\(…\)` subexpression. The exact form that the positions are returned in depends on the function originally used for the match. For example, with `string-match`, they are integers (zero-based indices into the string), but with `looking-at`, they're buffer markers. In all cases, `nil` is returned for parentheses that didn't exist or were not part of the match.

`match-beginning` and `match-end` are only convenient interfaces to `match-data`, which represents the location of the match and of all subexpression matches. It's a list like:

```
(   (match-beginning 0) (match-end 0)
    (match-beginning 1) (match-end 1)
    (match-beginning 2) (match-end 2)
       ⋮
)
```

Applying ⌜`a`**`\(`**`b`**`?\)`**`\(c\)`⌟ to '`ac`' with `string-match` results in `match-data` returning `(0 2 1 1 1 2)`. The middle `1 1` indicates that the first set of capturing parentheses, those wrapping ⌜`b?`⌟, matched no text (the starting and ending positions are the same), but that it did so successfully starting at string index `1`).

However, if matching ⌜`a`**`\(`**`b`**`\)?`**`\(c\)`⌟ instead (note the question mark has moved), `match-data` returns `(0 2 nil nil 1 2)`, the `nil nil` indicating that the subexpression within the first set of parentheses did not successfully participate in the match. (The question mark was successful, but the parentheses it governed were not.)

The `match-string` and `replace-match` functions use the information from `match-data` to retrieve or modify the matched text (or maybe something else, as we'll see in the next paragraph). The form **`(match-string num)`** returns the text in the current buffer from **`(match-beginning num)`** to **`(match-end num)`**. With **`(match-string num string)`**, a substring of the given *`string`* is returned.

It's up to you to use `match-string` and `replace-match` with the same target text as the match function used to fill `match-data`. Nothing stops you from modifying the target text between the match and a call to `match-string`, or from moving to a different buffer, or from supplying a different string. I suppose there could be interesting uses for shenanigans like this, but for the most part they're *gotchas* to be avoided.

Benchmarking in Emacs

Being a full text editor, benchmarking in Emacs is often more involved than with most utilities, but it certainly can be done. The listing on the next page shows one of the programs used to benchmark the example at the end of Chapter 5 (☞ 178). I'm not the most stellar of *elisp* programmers, so take it with a grain of salt. The `time-now` function should prove useful, though.

Emacs Regex Optimizations

For all its use of regular expressions, the Emacs regex engine is the least optimized of the advanced NFA tools mentioned in this book. Among the engine optimizations mentioned in Chapter 5 (☞ 154), it does only *first-character-discrimination, simple repetition,* and a half-hearted attempt at *length cognizance* (by noting that if the pattern cannot match an empty string, applying the pattern at the end of the target text can be skipped). It doesn't even optimize line anchors.

The *first character discrimination,* however, is the best among these NFA tools. Whereas other tools just throw up their hands with something as simple as ⌈`a|b`⌋, Emacs's optimization understands the full complexity of an expression. With ⌈`^[·TAB]*\(with\|pragma\|use\)`⌋, for example, it correctly realizes that a match must start with ⌈`[TAB·puw]`⌋. This goes a *long* way toward making Emacs regexes efficient enough to be reasonably usable to the extent they are today (example ☞ 178).

As explained in "Compile Caching" (☞ 158), a regex is normally compiled each time it is used, but Emacs caches the most-recently-used regexes. As of version

Code for Emacs function for Chapter 5 benchmark

```
  ;; -*- lisp-interaction -*-

  (defun time-now ()
    "Returns the current time as the floating-point number of seconds
  since 12:00 AM January 1970."
      (+ (car (cdr (current-time)))
         (/ (car (cdr (cdr (current-time)))) 1000000.0))
  )

  (defun dotest () "run my benchmark" (interactive)
    (setq case-fold-search t)       ;; Want case-insensitive matching.
    (goto-line 1)                   ;; Go to top of buffer.
    (setq count 0)                  ;; No matches seen so far.
    (message "testing... ")         ;; Let the user know we're working.
    (setq start (time-now))         ;; Note when we start.

    (while (< (point)  (point-max)) ;; So long as we're not at the end . . . .
      (setq beg (point))            ;; Note the beginning of the line.
      (forward-line 1)              ;; Go to next line.
      (setq end (point))            ;; Note start of next line.
      (goto-char beg)               ;; Move back to start of this line.

      (if
          ;; Try each match in turn
          (or
           (re-search-forward    "\\<char\\>"      end t)
           (re-search-forward    "\\<const\\>"     end t)
               _
           (re-search-forward    "\\<unsigned\\>"  end t)
           (re-search-forward    "\\<while\\>"     end t)
           )
          (setq count  (+ count 1))  ;;   note that we found another matching line.
       )
     (goto-char end)   ;;   move next
     )

   ;; all-done -- calculate and report how much time was taken
   (setq delta (- (time-now) start))
   (message "results: count is %d, time = %.2f sec" count delta)
)
```

19.33, it caches five, but it will cache 20 in a future version.* Increasing the cache size was extremely beneficial to the example benchmarked in Chapter 5 (☞ 178), but that test is pretty much a worst-case situation for the cache. I did some simple, real-world tests using automatic buffer indentation and fontification routines (which make heavy use of regular expressions), and found the larger cache size won about a 20 percent improvement. Still, 20 percent is nothing to sneeze at!

If you build your own copy of Emacs, you can set the cache size as you like. Just set `REGEXP_CACHE_SIZE` at the top of *src/search.c*.

---

* A few days before this book went to final copyedit, I told Richard Stallman about the benchmark in Chapter 5, and he decided to bump up the cache size to 20, and also made the cache search somewhat more efficient. These changes should appear in a future version of Emacs.

7

Perl Regular Expressions

In this chapter:
- ***The Perl Way***
- ***Regex-Related Perlisms***
- ***Perl's Regex Flavor***
- ***The Match Operator***
- ***The Substitution Operator***
- ***The Split Operator***
- ***Perl Efficiency Issues***
- ***Putting It All Together***
- ***Final Comments***

Perl has been featured prominently in this book, and with good reason. It is popular, extremely rich with regular expressions, freely and readily obtainable, easily approachable by the beginner, and available for a wide variety of platforms, including Amiga, DOS, MacOS, NT, OS/2, Windows, VMS, and virtually all flavors of Unix.

Some of Perl's programming constructs superficially resemble those of C or other traditional programming languages, but the resemblance stops there. The way you wield Perl to solve a problem *The Perl Way* is very different from traditional languages. The overall layout of a Perl program often uses the traditional structured and object-oriented concepts, but data processing relies *very* heavily on regular expressions. In fact, I believe it is safe to say that **regular expressions play a key role in virtually all Perl programs.** This includes everything from huge 100,000-line systems, right down to simple one-liners, like

```
  % perl -pi -e 's[(\d+(\.\d*)?)F\b]{sprintf "%.0fC", ($1-32) * 5/9}eg' *.txt
```

which goes through `*.txt` files and replaces Fahrenheit values with Celsius ones (reminiscent of the first example from Chapter 2).

In This Chapter

In this chapter we'll look at everything regex about Perl—the details of its regex flavor and the operators that put them to use. This chapter presents the regex-relevant details from the ground up, but I assume that you have at least a basic familiarity with Perl. (If you've read Chapter 2, you're probably already familiar enough to at least start using this chapter.) I'll often use, in passing, concepts that have not yet been examined in detail, and won't dwell much on non-regex aspects of the language. It might be a good idea to keep the Perl manpage handy, or perhaps O'Reilly's *Perl 5 Desktop Reference* (see Appendix A).

Perhaps more important than your current knowledge of Perl is your *desire to understand more.* This chapter is not light reading by any measure. Because it's not my aim to teach Perl from scratch, I am afforded a luxury that general books about Perl do not have: I don't have to omit important details in favor of weaving one coherent story that progresses unbroken through the whole chapter. What remains coherent throughout is the drive for a total understanding. Some of the issues are complex, and the details thick; don't be worried if you can't take it all in at once. I recommend first reading through to get the overall picture, and returning in the future to use as a reference as needed.

Ideally, it would be nice if I could cleanly separate the discussion of the regex flavor from the discussion on how to apply them, but with Perl the two are inextricably intertwined. To help guide your way, here's a quick rundown of how this chapter is organized:

- In "The Perl Way", I begin with an overall look at how Perl and regular expressions talk to each other and to the programmer. I present a short example that investigates how to solve a problem in *The Perl Way,* introducing some important Perl concepts that we'll encounter time and time again throughout this chapter and you'll encounter throughout your life with Perl.

- "Regex-Related Perlisms" (page 210) looks at some aspects of Perl that are particularly important to regular expressions. Concepts such as *dynamic scoping, expression context,* and *interpolation* are covered in detail with a strong bent toward their relationship with regular expressions.

- "Perl's Regex Flavor" (page 225) looks at the real McCoy, including all those tasty goodies that were new in Perl version 5. Not particularly useful without a way to apply them, "The Match Operator" (☞ 246), "The Substitution Operator" (☞ 255), and "The Split Operator" (☞ 259) provide all the details to Perl's sometimes magical regex controls.

- "Perl Efficiency Issues" (page 265) delves into an area close to every Perl programmer's heart. Perl uses a Traditional NFA match engine, so you can feel free to start using all the techniques from Chapter 5 right away. There are, of course, Perl-specific issues that can greatly affect how, and how quickly, Perl applies your regexes, and we'll look at them here.

- Finally, in "Putting It All Together" (page 290) I conclude by looking at a few examples to help put it all into perspective. I also re-examine the example from "The Perl Way" in light of all this chapter covers. The final example, matching an official Internet email address (☞ 294), brings to bear every ounce of technique we can muster. The result is a regex *almost 5,000 characters long,* yet it's one that we can understand and manage with confidence.

The Perl Way

Table 7-1 summarizes Perl's extremely rich regular-expression flavor. If you're new to Perl after having worked with another regex-endowed tool, many items will be unfamiliar. Unfamiliar, yes, but tantalizing! Perl's regular-expression flavor is perhaps the most powerful among today's popular tools. One key to more than a superficial understanding of Perl regular expressions is knowing that it uses a Traditional NFA regex engine. This means that all the NFA techniques described in the previous chapters can be brought to bear in Perl.

Yet, hacker does not live by metacharacters alone. Regular expressions are worthless without a means to apply them, and Perl does not let you down here either. In this respect, Perl certainly lives up to its motto "There's more than one way to do it."

Table 7-1: Overview of Perl's Regular-Expression Language

. (*dot*)	any byte except newline (☞ 233) (any byte *at all* with the /s modifier ◊ ☞234)		
|	alternation	`(…)`	normal grouping and capturing
greedy quantifiers (☞225)		`(?:…)`	pure grouping only◊ (☞227)
`*` `+` `?` `{n}` {*min*,}{*min*,*max*}		`(?=…)`	positive lookahead◊ (☞228)
non-greedy quantifiers◊ (☞225)		`(?!…)`	negative lookahead◊ (☞228)
`*?` `+?` `??` `{n}?` {*min*,}? {*min*,*max*}?		**anchors**	
`(?#…)`	comment◊ (☞230)	`\b`• `\B`	word/non-word anchors (☞240)
`#…`	(with `/x` mod, ☞223) comment† until newline or end of regex	`^` `$`	start/end of string (or start and end of logical line)◊ (☞232)
inlined modifiers◊ (☞ 231)		`\A` `\Z`	start/end of string◊ (☞234)
(?*mods*)	mods from among `i`, `x`, `m`, and `s`	`\G`	end of previous match◊ (☞236)
`\1`, `\ 2`, etc.	text previously matched by associated set of capturing parentheses (☞227)		
`[…]` `[^…]`	Normal and inverted character classes (☞243)		
	(the items below are also valid within a character class)		
character shorthands (☞241)		**class shorthands** (☞241)	
`\b`• `\t` `\n` `\r` `\f` `\a` `\e` \num \x *num* \c *char*		`\w` `\W` `\s` `\S` `\d` `\D`	
`\l` `\u` `\L` `\U` `\Q`◊ `\E`	on-the-fly text modification (☞245)		

◊ Not available before Version 5.000

† Not reliably available before Version 5.002

• \b is an anchor outside of a character class, a shorthand inside.

Regular Expressions as a Language Component

An attractive feature of Perl is that regular expression support is so deftly built-in as part of the language. Rather than providing stand-alone functions for applying regular expressions, Perl provides regular-expression *operators* that are meshed well with the rich set of other operators and constructs that make up the Perl language. Table 7-2 briefly notes the language components that have some direct connection with regular expressions.

Perhaps you never considered "`... =~ m/.../`" to be an operator, but just as addition's `+` is an operator which takes two operands and returns a sum, the match is an operator that takes two operands, a regex operand and a target-string operand, and returns a value. As discussed in Chapter 5's "Functions *vs.* Integrated Features *vs.* Objects" (☞ 159), the main difference between a function and an operator is that operators can treat their operands in magical ways that a function normally can't.* And believe me, there's lots of magic surrounding Perl's regex operators. But remember what I said in Chapter 1: There's nothing magic about magic if you *understand* what's happening. This chapter is your guide.

There are some not-so-subtle differences between a *regular expression* and a *regular-expression operand.* You provide a raw regex operands in your script, Perl cooks it a bit, then gives the result to the regex search engine. The preprocessing (cooking) is similar to, but not exactly the same as, what's done for doublequoted strings. For a superficial understanding, these distinctions are not important *which is why I'll explain and highlight the differences at every opportunity!*

Don't let the shortness of Table 7-2's "regex-related operators" section fool you. Each of those three operators really packs a punch with a variety of options and special-case uses.

Perl's Greatest Strength

The richness of variety and options among the operators and functions is perhaps Perl's greatest feature. They can change their behavior depending on the context in which they're used, often doing just what the author naturally intends in each differing situation. In fact, the Second Edition of O'Reilly's *Programming Perl* goes so far as to boldly state "In general, Perl operators do exactly what you want. . . ." The regex match operator `m/`*regex*`/`, for example, offers a wide variety of different functionality depending upon where, how, and with which modifiers it is used. The flexibility is amazing.

---

* The waters are muddied in Perl, where functions and procedures can also respond to their context, and can even modify their arguments. I try to draw a sharp distinction with the regex features because I wish to highlight characteristics which can easily lead to misunderstandings.

Table 7-2: Overview of Perl's Regex-Related Items

Regex-Related Operators	
** `m/regex/mods`	(☞246)
** `s/regex/subst/mods`	(☞255)
`split(...)`	(☞259)

**operates on `$_` unless related via `=~` or `!~`

modifier (☞249)	**modifies how. . .**
`/x`◦ `/o`	regex is interpreted
`/s`◦ `/m`◦ `/i`	engine considers target text
`/g` `/e`	other

Related Variables	
`$_`	default search target
`$*`	obsolete multi-line mode (☞232)

After-Match Variables (☞217)	
`$1, $2, etc.`	captured text
`$+`	highest filled `$1, $2, ...`
`` $` `` `$&` `$'`	text before, of, and after match

(best to avoid-see "Perl Efficiency Issues" ☞265)

—Related Functions—

`pos` (☞239)	`study` (☞287)	`quotemeta`	`lc`	`lcfirst`	`uc`	`ucfirst`	(☞245)

◦ Not available before Version 5.000

Perl's Greatest Weakness

This concentrated richness in expressive power is also one of Perl's least-attractive features. There are innumerable special cases, conditions, and contexts that seem to change out from under you without warning when you make a subtle change in your code—you've just hit another special case you weren't aware of.[*] The *Programming Perl* quote continues ". . . unless you want consistency." Certainly, when it comes to computer science, there is a certain art-like appreciation to boring, consistent, dependable interfaces. Perl's power can be a devastating weapon in the hands of a skilled user, but it seems with Perl, you become skilled by repeatedly shooting yourself in the foot.

In the Spring 1996 issue of *The Perl Journal*[**] Larry Wall wrote:

> One of the ideas I keep stressing in the design of Perl is that things that ARE different should LOOK different.

This is a good point, but with the regular expression operators, differences unfortunately aren't always readily apparent. Even skilled hackers can get hung up on the myriad of options and special cases. If you consider yourself an expert, don't tell me you've never wasted way too much time trying to understand why

```
if (m/.../g) {
      ⋮
```

wasn't working, because I probably wouldn't believe you. Everyone does it eventually. (If you don't consider yourself an expert and don't understand what's wrong with the example, have no fear: this book is here to change that!)

---

[*] That they're innumerable doesn't stop this chapter from trying!

[**] See `http://tpj.com/ or staff@tpj.com`

In the same article, Larry also wrote:

> In trying to make programming predictable, computer scientists have mostly succeeded in making it boring.

This is also true, but it struck me as rather funny since I'd written "there is a certain art-like appreciation to boring, consistent, dependable interfaces" for this section's introduction only a week before reading Larry's article! My idea of "art" usually involves more engineering than paint, so what do I know? In any case, I highly recommend Larry's entertaining and thought provoking article for some extremely insightful comments on Perl, languages, and yes, art.

A Chapter, a Chicken, and The Perl Way

Perl regular expressions provide so many interrelated concepts to talk about that it's the classic *chicken and egg* situation. Since the Chapter 2 introduction to Perl's approach to regular expressions was light, I'd like to present a meaty example to get the ball rolling before diving into the gory intricacies of each regex operator and metacharacter. I hope it will become familiar ground, because it discusses many concepts addressed throughout this chapter. It illustrates The Perl Way to approach a problem, points out pitfalls you might fall into, and may even hatch a few chickens from thin air.

An Introductory Example: Parsing CSV Text

Let's say that you have some `$text` from a CSV (Comma Separated Values) file, as might be output by dBASE, Excel, and so on. That is, a file with lines such as:

```
"earth",1,,"moon", 9.374
```

This line represents five fields. It's reasonable to want this information as an array, say `@field`, such that `$field[0]` was '`earth`', `$field[l]` was '`1`', `$field[2]` was undefined, and so forth. This means not only splitting the data into fields, but removing the quotes from quoted fields. Your first instinct might be to use `split`, along the lines of:

```
@fields = split(/,/, $text);
```

This finds places in `$text` where 「`,`」 matches, and fills `@fields` with the snippets that those matches delimit (as opposed to the snippets that the matches match).

Unfortunately, while `split` is quite useful, it is not the proper hammer for this nail. It's inadequate because, for example, matching only the comma as the delimiter leaves those doublequotes we want to remove. Using 「`"?,"?`」 helps to solve this, but there are still other problems. For example, a quoted field is certainly allowed to have commas *within* it, and we don't want to treat those as field delimiters, but there's no way to tell `split` to leave them alone.

Perl's full toolbox offers many solutions; here's one I've come up with:

```
    @fields = ();# initialize @fields to be empty
    while ($text =~ m/"([^"\\]*(\\.[^"\\]*)*)",?|([^,]+),?|,/g){
       push(@fields, defined($1) ? $1 : $3); # add the just-matched
field
    }
    push(@fields, undef) if $text =~  m/,$/;# account for an empty last
field
    # Can now access the data via @fields ...
```

Even experienced Perl hackers might need more than a second glance to fully grasp this snippet, so let's go over it a bit.

Regex operator context

The regular expression here is the somewhat imposing:

⌈`"([^"\\]*(\\.[^"\\]*)*)",?|([^,]+),?|,`⌋

Frankly, looking at the expression is not meaningful until you also consider how it is used. In this case, it is applied via the match operator, using the `/g` modifier, as the conditional of a `while`. This is discussed in detail in the meat of the chapter, but the crux of it is that the match operator behaves differently depending how and where it is used. In this case, the body of the `while` loop is executed once each time the regex matches in `$text`. Within that body are available any `$&`, `$1`, `$2`, etc., set by each respective match.

Details on the regular expression itself

That regular expression isn't really as imposing as it looks. From a top-level view, it is just three alternatives. Let's look at what these expressions mean from a local, standalone point of view. The three alternatives are:

⌈`"([^"\\]*(\\.[^"\\]*)*)",?`⌋

This is our friend from Chapter 5 to match a doublequoted string, here with ⌜`,?`⌟ appended. The marked parentheses add no meaning to the regular expression itself, so they are apparently used only to capture text to `$1`. This alternative obviously deals with doublequoted fields of the CSV data.

⌜`([^,]+),?`⌟

This matches a non-empty sequence of non-commas, optionally followed by a comma. Like the first alternative, the parentheses are used only to capture matched text—this time everything up to a comma (or the end of text). This alternative deals with non-quoted fields of the CSV data.

⌜`,`⌟

Not much to say here—it simply matches a comma.

These are easy enough to understand as individual components, except perhaps the significance of the ⌈`,?`⌋, which I'll get to in a bit. However, they do not tell us much individually   we need to look at how they are combined and how they work with the rest of the program.

How the expression is actually applied

Using the combination of `while` and `m/…/g` to apply the regex repeatedly, we want the expression to match once for each field of the CSV line. First, let's consider how it matches just the first time it's applied to any given line, as if the `/g` modifier were not used.

The three alternatives represent the three types of fields: quoted, unquoted, and empty. You'll notice that there's nothing in the second alternative to stop it from matching a quoted field. There's no need to disallow it from matching what the first alternative can match since Perl's Traditional NFA's non-greedy alternation guarantees the first alternative will match whatever it should, never leaving a quoted field for the second alternative to match against our wishes.

You'll also notice that by whichever alternative the first field is matched, the expression always matches through to the field-separating comma. This has the benefit of leaving the *current position* of the `/g` modifier right at the start of the next field. Thus, when the `while` plus `m/…/g` combination iterates and the regular expression is applied repeatedly, we are always sure to begin the match at the start of a field. This "keeping in synch" concept can be quite important in many situations where the `/g` modifier is used. In fact, it is exactly this reason why I made sure the first two alternatives ended with ⌈`,?`⌋. (The question mark is needed because the final field on the line, of course, will not have a trailing comma.) We'll definitely be seeing this keeping-in-synch concept again.

Now that we can match each field, how do we fill `@fields` with the data from each match? Let's look at `$1` and such. If the field is quoted, the first alternative matches, capturing the text within the quotes to `$1`. However, if the field is unquoted, the first alternative fails and the second one matches, leaving `$1` undefined and capturing the field's text to `$3`. Finally, on an empty field, the third alternative is the one that matches, leaving both `$1` and `$3` undefined. This all crystallizes to:

```
    push(@fields, defined($1) ? $1 : $3);
```

The marked section says "Use `$1` if it is defined, or `$3` otherwise." If neither is defined, the result is `$3`'s `undef`, which is just what we want from an empty field. Thus, in all cases, the value that gets added to `@fields` is exactly what we desire, and the combination of the `while` loop, the `/g` modifier, and keeping-in-synch allows us to process all the fields.

Well, almost all the fields. If the last field is empty (as demonstrated by a line-ending comma), our program accounts for it not with the main regex, but after the fact with a separate line to add an `undef` to the list. In these cases, you might think that we can just let the main regular expression match the nothingness at the end of the line. This would work for lines that do end with an empty field, but would tack on a phantom empty field to lines that don't, since *all* lines have nothingness at the end, so to speak.

Regular Expressions and The Perl Way

Despite getting just a bit bogged down with the discussion of the particular regular expression used, I believe that this has served as an excellent example for introducing regular expressions and The Perl Way:

- Judging from traffic in the Perl newsgroups and my private mail, parsing CSV data is a fairly common task. It is a problem that would likely be tackled by using brute force character-by-character analysis with languages such as C and Pascal, yet in Perl is best approached quite differently.

- It is not a problem that is solved simply with a single regular-expression match or regex search-and-replace, but one that brings to bear various intertwined features of the language. The problem first appears that it might call for `split`, but upon closer examination, that avenue turns out to be a dead end.

- It shows some potential pitfalls when the use of regular expressions is so closely knit with the rest of the processing. For example, although we would like to consider each of the three alternatives almost independently, the knowledge that the first alternative has two sets of parentheses must be reflected by the use of `$3` to access the parentheses of the second alternative. A change in the first alternative that adds or subtracts parentheses must be reflected by changing all related occurrences of `$3`—occurrences which could be elsewhere in the code some distance away from the actual regular expression.

- It's also an example of reinventing the wheel. The standard Perl (Version 5) library module `Text::ParseWords` provides the quotewords routine,* so:

```
use Text::ParseWords;
       ⋮
@fields = quotewords(',', 0, $text);
```

is all you really need.

---

* Not to dilute my point, but I should point out that at the time of this writing, there are bugs in the `quotewords` routine. Among them, it strips the final field if it is the number zero, does not recognize trailing empty lines, and does not allow escaped items (except escaped quotes) within quoted fields. Also, it invokes an efficiency penalty on the whole script (☞ 277). I have contacted the author, and these concerns will hopefully be fixed in a future release.

> Of course, being able to solve a problem by hand is a good skill to have, but when efficiency isn't at an absolute premium, the readability and maintainability of using a standard library function is most appealing. Perl's standard library is extensive, so getting to know it puts a huge variety of both high- and low-level functionality at your fingertips.

Perl Unleashed

Larry Wall released Perl to the world in December 1987, and it has been continuously updated since. Version 1 used a regular-expression engine based upon Larry's own news browser *rn,* which in turn was based upon the regex engine in James Gosling's version of Emacs (the first Emacs for Unix). Not a particularly powerful regex flavor, it was replaced in Version 2 by an enhanced version of Henry Spencer's widely used regular-expression package. With a powerful regular expression flavor at its disposal, the Perl's language and regular expression features soon became intertwined.

Version 5, Perl5 for short, was officially released in October 1994 and represented a major upgrade. Much of the language was redesigned, and many regular expression features were added or modified. One problem Larry Wall faced when creating the new features was the desire for backward compatibility. There was little room for Perl's regular expression language to grow, so he had to fit in the new notations where he could. The results are not always pretty, with many new constructs looking ungainly to novice and expert alike. Ugly yes, but as we will see, *extremely* powerful.

Perl4 vs. Perl5

Because so many things were updated in version 5.000, many continued to use the long-stable version 4.036 (Perl4 for short). Despite Perl5's maturity, at the time of this writing Perl4 still remains, perhaps unfortunately,* in use. This poses a problem for someone (at the time of this writing, me) writing about Perl. You can write code that will work with either version, but doing so causes you to lose out on many of the great features new with Perl5. For example, a modern version of Perl allows the main part of the CSV example to be written as:

```
    push(@fields, $+) while $text =~ m{
        "([^\"\\]*(?:\\.[^\"\\]*)*)",?    #  Standard quoted string (with
possible comma)
      | ([^,]+)?                          #  or   up to next comma (with
possible comma)
      | ,                                 #  or   just a comma.
    }gx;
```

---

* Tom Christiansen suggested that I use "dead flea-bitten camel carcasses" instead of "Perl4, to highlight that anything before version 5 is dead and has been abandoned by most. I was tempted.

Those comments are actually part of the regular expression . . . much more readable, don't you think? As we'll see, various other features of Perl5 make this solution more appealing—we'll re-examine this example again in "Putting It All Together" (☞ 290) once we've gone over enough details to make sense of it all.

I'd like to concentrate on Perl5, but ignoring Perl4 ignores a lingering reality. I'll mention important Perl4 regex-related differences using markings that look like this: *[1 ☞305]*. This indicates that the first Perl4 note, related to whatever comment was just made, can be found on page 305. Since Perl4 is so old, I won't feel the need to recap everything in Perl4's manpage—for the most part, the Perl4 notes are short and to the point, targeting those unfortunate enough to have to maintain code for both versions, such as a site's Perlmaster. Those just starting out with Perl should definitely not be concerned with an old version like Perl4.

Perl5 vs. Perl5

To make matters more complex, during the heady days surrounding Perl5's early releases, discussions on what is now USENET's `comp.lang.perl.misc` resulted in a number of important changes to the language and to its regular expressions. For example, one day I was responding to a post with an extraordinarily long regular expression, so I "pretty-printed" it, breaking it across lines and otherwise adding whitespace to make it more readable. Larry Wall saw the post, thought that one really should be able to express regexes that way, and right there and then added Perl5's `/x` modifier, which causes most whitespace to be ignored in the associated regular expression.

Around the same time, Larry added the ⌈`(?#...)`⌋ construct to allow comments to b*e* embedded within a regex. A few months later, though, after discussions in the newsgroup, a raw '#' was also made to start a comment if the `/x` modifier were used. This appeared in version 5.002.* There were other bug fixes and changes as well—if you are using an early release, you might run into incompatibilities as you follow along in this book. I recommend version 5.002 or later.

As this second printing goes to press, version 5.004 is about to hit beta, with a final release targeted for Spring 1997. Regex-related changes in the works include enhanced locale support, modified `pos` support, and a variety of updates and optimizations. (For example, Table 7-10 will likely become almost empty.) Once 5.004 is officially released, this book's home page (see Appendix A) will note the changes.

---

* Actually, it appeared in some earlier versions, but did not work reliably.

Regex-Related Perlisms

There are a variety of concepts that are part of Perl in general, but are of particular interest to our study of regular expressions. The next few sections discuss:

- **context** An important Perl concept is that many functions and operators respond to the *context* they're used in. For instance, Perl expects a scalar value as the conditional of the `while` loop the main match operator in the CSV example would have behaved quite differently had it been used where Perl was expecting a list.

- **dynamic scope** Most programming languages support the concept of local and global variables, but Per! provides an additional twist with something known as *dynamic scoping*. Dynamic scoping temporarily "protects" a global variable by saving a copy and automatically restoring it later. It's an intriguing concept that's important for us because it affects `$1` and other match-induced side effects.

- **string processing** Perhaps surprising to those familiar only with traditional programming languages like C or Pascal, string "constants" in Perl are really highly functional and dynamic operators. You can even call a function from within a string! Perl regular expressions usually receive very similar treatment, and this creates some interesting effects that we'll investigate.

Expression Context

The notion of *context* is important throughout the Perl language, and in particular, to the match operator. In short, an expression might find itself in one of two contexts: *A list context** is one where a list of values is expected, while a *scalar context* is where a single value is expected.

Consider the two assignments:

```
$s = expression one;
@a = expression two;
```

Because `$s` is a simple scalar variable (holds a single value, not a list), it expects a simple scalar value, so the first expression, whatever it may be, finds itself in a scalar context. Similarly, because `@a` is an array variable and expects a list of values, the second expression finds itself in a list context. Even though the two expressions might be exactly the same, they might (depending on the case) return completely different values, and cause completely different side effects while they're at it.

---

* "List context" used to be called "array context". The change recognizes that the underlying data is a list that applies equally well to an `@array`, a `%hash`, and (`an`, `explicit`, `list`).

Sometimes, the type of an expression doesn't exactly match the type of value expected of it, so Perl does one of two things to make the square peg fit into a round hole: *1)* it allows the expression to respond to its context, returning a type appropriate to the expectation, or *2)* it contorts the value to make it fit.

Allowing an expression to respond to its context

A simple example is a file I/O operator, such as `<MYDATA>`. In a list context, it returns a list of all (remaining) lines from the file. In a scalar context, it simply returns the next line.

Many Perl constructs respond to their context, and the regex operators are no different. The match operator `m/…/`, for example, sometimes returns a simple true/false value, and sometimes a list of certain match results. All the details are found later in this chapter.

Contorting an expression

If a list context still ends up getting a scalar value, a list containing the single value is made on the fly. Thus, **`@a = 42`** is the same as **`@a = (42)`**. On the other hand, there's no general rule for converting a list to a scalar. If a literal list is given, such as with

```
    $var = ($this, &is, 0xA, 'list');
```

the comma-operator returns the last element, `'list'`, for `$var`. If an array is given, as with **`$var = @array`**, the length of the array is returned.

Some words used to describe how other languages deal with this issue are *cast, promote, coerce,* and *convert,* but I feel they are a bit too consistent (boring?) to describe Perl's attitude in this respect.

Dynamic Scope and Regex Match Effects

Global and private variables

On a broad scale, Perl offers two types of variables: global and private.* Not available before Perl5, private variables are declared using `my(...)`. Global variables are not declared, but just pop into existence when you use them. Global variables are visible from anywhere within the program, while private variables are visible, lexically, only to the end of their enclosing block. That is, the only Perl code that can access the private variable is the code that falls between the `my` declaration and the end of the block of code that encloses the `my`.

---

* Perl allows the names of global variables to be partitioned into groups called *packages,* but the variables are still global.

Dynamically scoped values

Dynamic scoping is an interesting concept that many programming languages don't provide. We'll see the relevance to regular expressions soon, but in a nutshell, you can have Perl save a copy of a global variable that you intend to modify, and restore the original copy automatically at the time when the enclosing block ends. Saving a copy is called *creating a new dynamic scope.* There are a number of reasons you might want to do this, including:

- Originally, Perl did not provide true local variables, only global variables. Even if you just needed a temporary variable, you had no choice but to use a global, and so risked stepping on someone else's toes if your choice for a temporary variable was unlucky enough to conflict with what was being used elsewhere. Having Perl save and restore a copy of your intended variable isolated your changes of the global variable, in time, to your temporary use.

- If you use global variables to represent "the current state", such as the name of a file being processed, you might want to have a routine modify that state only temporarily. For example, if you process a file-include directive, you want the "current filename" to reflect the new file only until it's done, reverting back to the original when the directive has been processed. Using dynamic scoping, you can have the save-and-restore done automatically.

The first use listed is less important these days because Perl now offers truly local variables via the `my` directive. Using `my` creates a new variable completely distinct from and utterly unrelated to any other variable anywhere else in the program. Only the code that lies between the `my` to the end of the enclosing block can have direct access to the variable.

The extremely ill-named function local creates a new dynamic scope. Let me say up front that *the call to `local` does not create a new variable.* Given a global variable, local does three things:

1. saves an internal copy of the variable's value; and

2. copies a new value into the variable (either `undef`, or a value assigned to the `local`); and

3. slates the variable to have the original value restored when execution runs off the end of the block enclosing the `local`.

This means that "local" refers only to how long any changes to the variable will last. The global variable whose value you've copied is still visible from anywhere within the program—if you make a subroutine call after creating a new dynamic scope, that subroutine (wherever it might be located in the script) will see any changes you've made. This is just like any normal global variable. The difference here is that when execution of the enclosing block finally ends, the previous value is automatically restored.

An automatic save and restore of a global variable's value—that's pretty much all there is to `local`. For all the misunderstanding that has accompanied `local`, it's no more complex than the snippet on the right of Table 7-3 illustrates.

Table 7-3: The meaning of local

Normal Perl

```
{
    local($SomeVar); # save copy
    $SomeVar = 'My Value';
        ⋮
        ⋮
        ⋮
} # automatically restore $SomeVar
```

Equivalent Meaning

```
{
     my $TempCopy = $SomeVar;
     $SomeVar = undef;
     $SomeVar = 'My Value';
        ⋮
     $SomeVar = $TempCopy;
}
```

(As a matter of convenience, you can assign a value to `local($SomeVar)`, which is exactly the same as assigning to `$SomeVar` in place of the `undef` assignment. Also, the parentheses can be omitted to force a scalar context.)

References to `$SomeVar` while within the block, or within a subroutine called from the block, or within a signal handler invoked while within the block—any reference from the *time* the `local` is called to the *time* the block is exited—references '`My Value`'. If the code in the block (or anyone else, for that matter) modifies `$Somevar`, everyone (including the code in the block) sees the modification, but it is lost when the block exits and the original copy is automatically restored.

As a practical example, consider having to call a function in a poorly written library that generates a lot of `Use of uninitialized value` warnings. You use Perl's `-w` option, as all good Perl programmers should, but the library author apparently didn't. You are exceedingly annoyed by the warnings, but if you can't change the library, what can you do short of stop using `-w` altogether? Well, you could set a local value of `$^W`, the in-code debugging flag (the variable name `^W` can be either the two characters, caret and '`W`', or an actual control-`W` character):

```
{
    local $^W = 0; # ensure debugging is off.
    &unruly_function(...);
}
# exiting the block restores the original value of $^W
```

The call to `local` saves an internal copy of the previous value of the global variable `$^W`, whatever it might have been. Then that same `$^W` receives the new value of zero that we immediately scribble in. When `unruly_function` is executing, Perl checks `$^W` and sees the zero we wrote, so doesn't issue warnings. When the function returns, our value of zero is still in effect.

So far, everything appears to work just as if you didn't use `local`. However, when the block is exited right after the subroutine returns, the saved value of `$^w` is restored. Your change of the value was local, in *time,* to the lifetime of the block.

You'd get the same effect by making and restoring a copy yourself, as in Table 7-3, but `local` conveniently takes care of it for you.

For completeness, let's consider what happens if I use `my` instead of `local`.* Using my creates a *new variable* with an initially undefined value. It is visible only within the lexical block it is declared in (that is, visible only by the code written between the my and the end of the enclosing block). It does not change, modify, or in any other way refer to or affect other variables, including any global variable of the same name that might exist. The newly created variable is not visible elsewhere in the program, including from within `unruly_function`. In our example snippet, the new `$^W` is immediately set to zero but is never again used or referenced, so it's pretty much a waste of effort. (While executing `unruly_function` and deciding whether to issue warnings, Perl checks the unrelated global variable `$^W`.)

A better analogy: clear transparencies

A useful analogy for `local` is that it provides a clear transparency over a variable on which you scribble your own changes. You (and anyone else that happens to look, such as subroutines and interrupt handlers) will see the new values. They shadow the previous value until the point in time that the block is finally exited. At that point, the transparency is automatically removed, in effect, removing any changes that might have been made since the `local`.

This analogy is actually much closer to reality than the original "an internal copy is made" description. Using `local` doesn't have Perl actually make a copy, but instead puts your new value earlier in the list of those checked whenever a variable's value is accessed (that is, it shadows the original). Exiting a block removes any shadowing values added since the block started. Values are added manually, with `local`, but some variables have their values automatically dynamically scoped. Before getting into that important regex-related concern, I'd like to present an extended example illustrating manual dynamic scoping.

An extended dynamic-scope example

As an extended, real life, *Perl Way* example of dynamic scoping, refer to the listing on the next page. The main function is `ProcessFile`. When given a filename, it opens it and processes commands line by line. In this simple example, there are only three types of commands, processed at 6, 7, and 8. Of interest here are the global variables `$filename`, `$command`, `$.`, and `%HaveRead`, as well as the global filehandle `FILE`. When `ProcessFile` is called, all but `%HaveRead` have their values dynamically scoped by the `local` at 3.

---

* Perl doesn't allow the use of `my` with this special variable name, so the comparison is only academic.

Dynamic Scope Example

```
    #  Process "this" command
    sub DoThis   1
    {
       print "$filename line $.: processing $command";
          ⋮
    }

    #  Process "that" command
    sub DoThat 2
    }
       print "$filename line $.: processing $command";
          ⋮
    }

    #  Given a filename, open file and process commands
    sub ProcessFile
    {
       local($filename) = @_;
       local(*FILE, $command, $.);3

       open(FILE, $filename)  || die qq/can't open "$filename":
$!\n/;

       $HaveRead{$filename) = 1; 4

       while ($command = <FILE>)
       {
           if ($command =~ m/^#include "(.*)"$/) {
              if (defined $HaveRead{$1}) {           5
                  warn qq/$filename $.: ignoring repeat include
of "$1"\n/;
              } else {
                  ProcessFile($1); 6
              }
           } elsif ($command =~ m/^do-this/) {
              DoThis; 7
           } elsif ($command =~ m/^do-that/) {
              DoThat; 8
           } else {
              warn "$filename $.: unknown command: $command";
          }
       }
       close(FILE);
    } 9
```

When a `do-this` command is found (7), the `DoThis` function is called to process it. You can see at 1 that the function refers to the global variables `$filename`, `$.`, and `$command`. The `DoThis` function doesn't know (nor care), but the values of these variables that it sees were written in `ProcessFile`.

The `#include` command's processing begins with the filename being plucked from the line at 5. After making sure the file hasn't been processed already, we call `ProcessFile` recursively, at 6. With the new call, the global variables `$filename`, `$command`, and `$.`, as well as the filehandle `FILE`, are again overlaid with a transparency that is soon updated to reflect the status and commands of the second file. When commands of the new file are processed within `ProcessFile` and the two subroutines, `$filename` and friends are visible, just as before.

Nothing at this point appears to be different from straight global variables.

The benefits of dynamic scoping are apparent when the second file has been processed and the related call of `ProcessFile` exits. When execution falls off the block at 9, the related `local` transparencies laid down at 3 are removed, restoring the original file's values of `$filename` and such. This includes the filehandle `FILE` now referring to the first file, and no longer to the second.

Finally, let's look at `%HaveRead`, used to keep track of files we've seen (4 and 5). It is specifically *not* dynamically scoped because we really do need it to be global across the entire time the script runs. Otherwise, included files would be forgotten each time `ProcessFile` exits.

Regex side-effects and dynamic-scoping

What does all this about dynamic scope have to do with regular expressions? A lot. Several variables are automatically set as a side effect of a successful match. Discussed in detail in the next section, they are variables like `$&` (refers to the text matched) and `$1` (refers to the text matched by the first parenthesized subexpression). These variables have their value *automatically* dynamically scoped upon entry to *every* block.

To see the benefit of this, realize that each call to a subroutine involves starting a new block. For these variables, that means a new dynamic scope is created. Because the values before the block are restored when the block exits (that is, when the subroutine returns), the subroutine can't change the values that the caller sees.

As an example, consider:

```
    if (m/(...)/)
    {
        &do_some_other_stuff();
        print "the matched text was $1.\n";
    }
```

Because the value of `$1` is dynamically scoped automatically upon entering each block, this code snippet neither cares, nor needs to care, whether the function `do_some_other_stuff` changes the value of `$1` or not. Any changes to `$1` by the function are contained within the block that the function defines, or perhaps within a sub-block of the function. Therefore, they can't affect the value this snippet sees with the `print` after the function returns.

The automatic dynamic scoping can be helpful even when not so apparent:

```
if ($result =~ m/ERROR=(.*)/) {
   warn "Hey, tell $Config{perladmin} about $1!\n";
}
```

(The standard library module `Config` defines an associative array `%Config`, of which the member `$Config{perladmin}` holds the email address of the local

Perlmaster.) This code could be very surprising if `$1` were not automatically dynamically scoped. You see, `%Config` is actually a tied variable, which means that any reference to it involves a behind-the-scenes subroutine call. `Config`'s subroutine to fetch the appropriate value when `$Config{…}` is used invokes a rege***x*** match. It lies between your match and your use of `$1`, so it not being dynamically scoped would trash the `$1` you were about to use. As it is, any changes in the `$Config{…}` subroutine are safely hidden by dynamic scoping.

Dynamic scoping vs. lexical scoping

Dynamic scoping provides many rewards if used effectively, but haphazard dynamic scoping with `local` can create a maintenance nightmare. As I mentioned, the `my(…)` declaration creates a private variable with *lexical scope.* A private variable's lexical scope is the opposite of a global variable's global scope, but it has little to do with dynamic scoping (except that you can't `local` the value of a my variable). Remember, `local` is an *action,* while `my` is an action *and* a declaration.

Special Variables Modified by a Match

A successful match or substitution sets a variety of global, automatically-dynamic-scoped, read-only variables._[1☞305]_* These values *never* change when a match attempt is unsuccessful, and are *always* set when a match is successful. As the case may be, they may be set to the empty string (a string with no characters in it), or undefined (a "no value here" value very similar to, yet testably distinct from, an empty string). In all cases, however, they are indeed set.

$& A copy of the text successfully matched by the regex. This variable (along with `` $` `` and `$'` below) is best avoided. (See "Unsociable $& and Friends" on page 273.) $& is never undefined after a successful match.

`` $` `` A copy of the target text in front of (to the left of) the match's start. When used in conjunction with the `/g` modifier, you sometimes wish `` $` `` to be the text from start of the *match attempt,* not the whole string. Unfortunately, it doesn't work that way.*[2☞305]* If you need to mimic such behavior, you can try using ⌈`\G([\x00-\xff]*?)`⌋ at the front of the regex and then refer to `$1`. `` $` `` is never undefined after a successful match.

`$'` A copy of the target text after (to the right of) the successfully matched text. After a successful match, the string ``"$`$&$'"`` is always a copy of the original target text.** `$'` is never undefined after a successful match.

---

* As described on page 209, this *[1☞305]*notation means that Perl4 note #1 is found on page 305.

** Actually, if the original target is undefined, but the match successful (unlikely, but possible), ``"$`$&$'"`` would be an empty string, not undefined. This is the only situation where the two differ.

`$1`, `$2`, `$3`, etc.

The text matched by the 1st, 2nd, 3rd, etc., set of capturing parentheses. (Note that `$0` is not included here—it is a copy of the script name and not related to regexes). These are guaranteed to be undefined if they refer to a set of parentheses that doesn't exist in the regex, or to a set that wasn't actually involved in the match.

These variables are available after a match, including in the replacement operand of `s/…/…/`. But it makes no sense to use them within the regex itself. (That's what ⌈`\1`⌋ and friends are for.) See "Using `$1` Within a Regex?" on the next page.

The difference between ⌈`(\w+)`⌋ and ⌈`(\w)+`⌋ can be seen in how these variables are set. Both regexes match exactly the same text, but they differ in what is matched within the parentheses. Matching against the string `tubby`, the first results in `$1` having `tubby`, while the latter in it having `y`: the plus is outside the parentheses, so each iteration causes them to start capturing anew.

Also, realize the difference between ⌈`(x)?`⌋ and ⌈`(x?)`⌋. With the former, the parentheses and what they enclose are optional, so `$1` would be either x or undefined. But with ⌈`(x?)`⌋, the parentheses enclose a match—what is optional are the contents. If the overall regex matches, the contents matches something, although that something might be the nothingness ⌈`x?`⌋ allows. Thus, with ⌈`(x?)`⌋ the possible values of `$1` are `x` and an empty string.

Perl4 and Perl5 treat unusual cases involving parentheses and iteration via star and friends slightly differently. It shouldn't matter to most of you, but I should at least mention it. Basically, the difference has to do with what `$2` will receive when something like ⌜`(main(OPT)?)+`⌟ matches only ⌜`main`⌟ and not ⌜`OPT`⌟ during the last successful iteration of the plus. With Perl5, because ⌜`(OPT)`⌟ did not match during the last successful match of its enclosing subexpression, `$2` becomes (rightly, I think) undefined. In such a case, Perl4 leaves `$2` as what it had been set to the last time ⌜`(OPT)`⌟ actually matched. Thus, with Perl4, `$2` is `OPT` if it had matched any time during the overall match.

`$+` A copy of the highest numbered `$1`, `$2`, etc. explicitly set during the match. If there are no capturing parentheses in the regex (or none used during the match), it becomes undefined.*[3☞306]* However, Perl does not issue a warning when an undefined `$+` is used.

When a regex is applied repeatedly with the `/g` modifier, each iteration sets these variables afresh. This is why, for instance, you can use `$1` within the replacement operand of `s/…/…/g` and have it represent a new slice of text each time. (Unlike the regex operand, the replacement operand *is* re-evaluated for each iteration; ☞ 255.)

Using `$1` within a regex?

The Perl manpage makes a concerted effort to point out that ⌜`\1`⌟ is not available as a backreference outside of a regex. (Use the variable `$1` instead.) `\1` is much more than a simple notational convenience—the variable `$1` refers to a string of static text matched during some previously completed successful match. On the other hand, ⌜`\1`⌟ is a true regex metacharacter to match text similar to that matched within the first parenthesized subexpression *at the time that the regex-directed NFA reaches the* ⌜`\1`⌟. What ⌜`\1`⌟ matches might change over the course of an attempt as the NFA tracks and backtracks in search of a match.

A related question is whether `$1` is available within a regex operand. The answer is "Yes, but not the way you might think." A `$1` appearing in a regex operand is treated exactly like any other variable: its value is interpolated (the subject of the next section) before the match or substitution operation even begins. Thus, as far as the regex is concerned, the value of `$1` has nothing to do with the current match, but remains left over from some previous match somewhere else.

In particular, with something like `s/…/…g`, the regex operand is evaluated, compiled once (also discussed in the next section), and then used by all iterations via `/g`. This is exactly opposite of the replacement operand, which is re-evaluated after each match. Thus, a `$1` within the replacement operand makes complete sense, but in a regex operand it makes virtually none.

"Doublequotish Processing" and Variable Interpolation

Strings as operators

Most people think of strings as *constants,* and, in practice a string like

```
    $month = "January";
```

is just that. `$month` gets the same value each time the statement is executed because `"January"` never changes. However, Perl can *interpolate* variables within doublequoted strings (that is, have the variable's value inserted in place of its name). For example, in

```
$message = "Report for $month:";
```

the value that `$message` gets depends on the value of `$month`, and in fact potentially changes each time the program does this assignment. The doublequoted string **`"Report for $month:"`** is exactly the same as:

```
'Report for ' . $month . ':'
```

(In a general expression, a lone period is Perl's string-concatenation operator; concatenation is implicit within a doublequoted string.)

The doublequotes are really *operators* that enclose operands. A string such as

```
"the month is $MonthName[&GetMonthNum]!"
```

is the same as the expression

```
'the month is ' . $MonthName[&GetMonthNum] . '!'
```

including the call to `GetMonthNum` *each time* the string is evaluated.*[4☞306]* Yes, you really can call functions from within doublequoted strings because doublequotes are operators! To create a true constant, Perl provides singlequoted strings, used here in the code snippet equivalences.

One of Perl's unique features is that a doublequoted string doesn't have to actually be delimited by doublequotes. The `qq/…/` notation provides the same functionality as `"…"`, so **`qq/Report for $month:/`** is a doublequoted string. You can also choose your own delimiters. The following example uses `qq{…}` to delimit the doublequoted string:

```
warn qq{"$ARGV" line $.: $ErrorMessage\n};
```

Singlequoted strings use `q/…/`, rather than `qq/…/`. Regular expressions use `m/…/` and `s/…/…/` for match and substitution, respectively. The ability to pick your own delimiters for regular expressions, however, is not unique to Perl: *ed* and its descendants have supported it for over 25 years.

Regular expressions as strings, strings as regular expressions

All this is relevant to regular expressions because the regex operators treat their regex operands pretty much (but not exactly) like doublequoted strings, including support for variable interpolation:

```
$field = "From";
     ⋮
if ($headerline =~ m/^$field:/) {
     ⋮
}
```

The marked section is taken as a variable reference and is replaced by the variable's value, resulting in ⌈`^From:`⌋ being the regular expression actually used.⌈`^$field:`⌋ is the regex operand; ⌈`^From:`⌋ is the actual regex after cooking. This seems similar to what happens with doublequoted strings, but there are a few differences (details follow shortly), so it is often said that it receives "doublequotish" processing.

One might view the mathematical expression **`($F - 32) * 5/9`** as

> *Divide*(*Multiply*(*Subtract*(`$F`, `32`), `5`), `9`)

to show how the evaluation logically progresses. It might be instructive to see **`$headerline =~ m/^$field:/`** presented in a similar way:

> *RegexMatch* (`$headerline`, *DoubleQuotishProcessing* ("`^$field:`"))

Thus, you might consider `^$field:` to be a match operand only indirectly, as it must pass through doublequotish processing first. Let's look at a more involved example:

```
    $single = qq{'([^'\\\\]*(?:\\\\.[^'\\\\]*)*)'}; #  to match a singlequoted string
    $double = qq{"([^"\\\\]*(?:\\\\.[^"\\\\]*)*)"}; #  to match a doublequoted string
    $string = "(?:$single|$double)";   #  to match either kind of string
           ⋮
    while (<CONFIG>){
        if (m/^name=$string$/o){
            $config{name} = $+;
        } else {
           ⋮
```

This method of building up the variable `$string` and then using it in the regular expression is much more readable than writing the whole regex directly:

```
    if (m/^name=(?:'[^'\\]*(?:\\.[^'\\]*)*'|"[^"\\]*)*")$/o) {
```

Several important issues surround this method of building regular expressions within strings, such as all those extra backslashes, the /o modifier, and the use of `(?...)` non-capturing parentheses. See "Matching an Email Address" (☞ 294) for a heady example. For the moment, let's concentrate on just how a regex operand finds its way to the regex search engine. To get to the bottom of this, let's follow along as Perl parses the program snippet:

```
    $header =~ m/^\Q$dir\E #  base directory
                  \/         #  separating slash
                  (.*)       #  grab the rest of the filename
                 /xgm;
```

For the example, we'll assume that `$dir` contains '`~/.bin`'.

The `\Q…\E` that wraps the reference to `$dir` is a feature of doublequotish string processing that is particularly convenient for regex operands. It puts a backslash in front of most symbol characters. When the result is used as a regex, it matches the literal text the `\Q…\E` encloses, even if that literal text contains what would otherwise be considered regex metacharacters (with our example of '`~/.bin`', the first three characters are escaped, although only the dot requires it).

Also pertinent to this example are the free whitespace and comments within the regex. Starting with Perl version 5.002, a regex operand subject to the `/x` modifier can use whitespace liberally, and may have raw comments. Raw comments start with # and continuing to the next newline (or to the end of the regex). This is but one way that doublequotish processing of regex operands differs from real doublequoted strings: the latter has no such `/x` modifier.

Figure 7-1 on page 223 illustrates the path from unparsed script, to regex operand, to real regex, and finally to the use in a search. Not all the phases are necessarily done at the same time. The lexical analysis (examining the script and deciding what's a statement, what's a string, what's a regex operand, and so on) is done just once when the script is first loaded (or in the case of `eval` with a string operand, each time the `eval` is re-evaluated). That's the first phase of Figure 7-1. The other phases can be done at different times, and perhaps even multiple times. Let's look at the details.

Phase A—identifying the match operand

The first phase simply tries to recognize the lexical extent of the regex operand. Perl finds `m`, the match operator, so it knows to scan a regex operand. At point *1* in the figure, it recognizes the slash as the operand delimiter, then searches for the closing delimiter, finding it at point *4*. For this first phase, strings and such get the same treatment, so there's no special regex-related processing here. One transformation that takes place in this phase is that the backslash of an escaped *closing* delimiter is removed, as at point *3.[5☞306]*

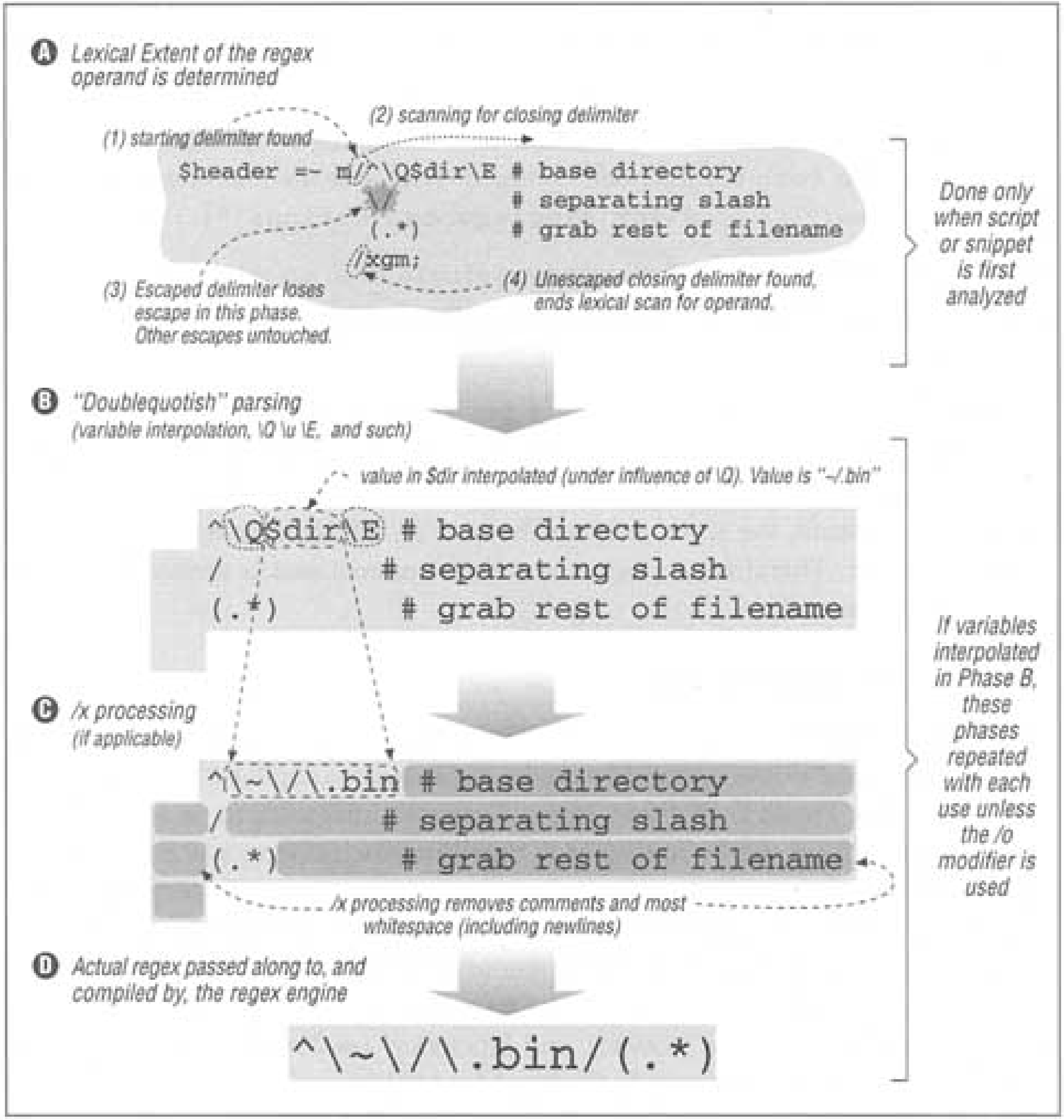
Phase B—doublequotish processing

The second phase of regex-operand parsing treats the isolated operand much like a doublequoted string. Variables are interpolated, and `\Q…\E` and such are processed. (The full list of these constructs is given in Table 7-8 on page 245.) With our example, the value of `$dir` is interpolated under the influence of `\Q`, so `'\~\/\.bin'` is actually inserted into the operand.

Although similar, differences between regex operands and doublequoted strings become apparent in this phase. Phase B realizes it is working with a regex operand, so processes a few things differently. For example, `\b` and `\3` in a doublequoted string always represent a backspace and an octal escape, respectively. But in a regex, they could also be the word-boundary metacharacter or a backreference, depending on where in the regex they're located—Phase B therefore leaves them undisturbed for the regex engine to later interpret as it sees fit. Another difference involves what is and isn't considered a variable reference. Something like `$|` will always be a variable reference within a string, but it is left for the regex engine to interpret as the metacharacters ⌈`$`⌋ and ⌈`|`⌋. Similarly, a string always interprets `$var[2-7]` as a reference to element `-5` of the array `@var` (which means the fifth element from the end), but as a regex operand, it is interpreted as a reference to `$var` followed by a character class. You can use the `${…}` notation for variable interpolation to force an array reference if you wish:

`$ {var[2-7]}.`

Due to variable interpolation, the result of this phase can depend on the value of variables (which can change as the program runs). In such a case, Phase B doesn't take place until the match code is reached during runtime. "Perl Efficiency Issues" (☞ 265) expands on this important point.


A
Lexical Extent of the regex operand is determined
(1) starting delimiter found
(2) scanning for closing delimiter
(3) Escaped delimiter loses escape in this phase. Other escapes untouched.
(4) Unescaped closing delimiter found, ends lexical scan for operand.
Done only when script or snippet is first analyzed
B
"Doublequotish" parsing (variable interpolation, \Q \u \E, and such)
value in $dir interpolated (under influence of \Q). Value is "~/.bin"
C
/x processing (if applicable)
/x processing removes comments and most whitespace (including newlines)
If variables interpolated in Phase B, these phases repeated with each use unless the /o modifier is used
D
Actual regex passed along to, and compiled by, the regex engine

```
$header =~ m/^\Q$dir\E # base directory
            \/         # separating slash
             (.*)      # grab rest of filename
            /xgm;
```

```
^\Q$dir\E # base directory
/        # separating slash
(.*)      # grab rest of filename
```

```
^\~\/\.bin # base directory
/          # separating slash
(.*)        # grab rest of filename
```

```
^\~\/\.bin/(.*)
```

Figure 7-1:
Perl parsing, from program text to regular expression engine

As discussed in the match and substitution operator sections later in this chapter, using a singlequote as the regex-operand delimiter invokes singlequotish processing. In such a case, this Phase B is skipped.

Phase C—/x processing

Phase C concerns only regex operands applied with `/x`. Whitespace (except within a character class) and comments are removed. Because this happens after Phase B's variable interpolation, whitespace and comments brought in from a variable end up being removed as well. This is certainly convenient, but there's one trap to watch out for. The # comments continue to the next newline or the end of the operand this is different from "until the end of an interpolated string." Consider if we'd included a comment at the end of page 221's `$single:`

```
    $single = qq{'(...regex here...)'  # for singlequoted
strings};
```

This value makes its way to `$string` and then to the regex operand. After Phase B, the operand is (with the intended comment bold, but the real comment underlined): ⌈`^name=(?:'(...)'` `•#•for•singlequoted•strings|"(...)")$`⌋

Surprise! The comment intended only for `$single` ends up wiping out what follows because we forgot to end the comment with a newline.

If, instead, we use:

```
    $single = qq{'(...regex here...)'   # for singlequoted
strings\n};
```

everything is fine because the `\n` is interpreted by the doublequoted string, providing an actual newline character to the regex when it's used. If you use a singlequoted `q{...}` instead, the regex receives the raw `\n`, which *matches* a newline, but *is not a* newline. Therefore, it doesn't end the comment and is removed with the rest of the comment.*

Phase D regex compilation

The result of Phase C is the true regular expression that the engine uses for the match. The engine doesn't apply it directly, but instead compiles it into an internal form. I call this Phase D. If there was no variable interpolation in Phase B, the same complied form can be reused each time the match operator is executed during the course of the program, and so each phase needs to be performed only once a real time savings. On the other hand, variable interpolation means that the regex can change each time the operator is reached, so Phases B through D must be performed again, each time. The important effects of this are the focus of "Regex Compilation, the `/o` Modifier, and Efficiency" (☞ 268). Also see the related discussion in Chapter 5's "Compile Caching" (☞ 158).

---

* Figure 7-1 is a model describing the complex multi-level parsing of Perl, but those digging around in Perl internals will find that the processing actually done by Perl is slightly different. For example, what I call Phase C is not a separate step, but is actually part of both Phases B and D.

I feel it is more clear to present it as a separate step, so have done so. (I spent a considerable amount of time coming up with the model that Figure 7-1 illustrates—I hope you'll find it helpful.) However, in the unlikely event of variable references within a comment, my model and reality could conceivably differ.

Consider: **`m/regex #comment $var/`**. In my model, `$var` is interpolated into the operand at Phase B, the results of which are removed with the rest of the comment in Phase C. In reality, the comment and variable reference are removed in Phase B before the variable is interpolated. The end results appear the same . . . almost always. If the variable in question contains a newline, that newline does *not* end the comment as my model would indicate. Also, any side effects of the variable interpolation, such as function calls and the like, are not done, since in reality the variable is not interpolated.

These situations are farfetched and rare, so they will almost certainly never matter to you, but I felt I should at least mention it.

Perl's Regex Flavor

Now that we've gotten some important tangential issues out of the way, let's look at Perl's regex flavor itself. Perl uses a Traditional NFA engine with a base set of metacharacters superficially resembling *egrep's,* but the resemblance to *egrep* stops there. The biggest differences are due to Perl's NFA, and to the many extra metacharacters that provide both convenient shorthands and additional raw power. The Perl manpage only summarizes Perl's regex flavor, so I will try to provide much more depth.

Quantifiers   Greedy and Lazy

Perl provides normal greedy quantifiers, with Perl5 adding the non-greedy counterparts as mentioned in Chapter 4. Greedy quantifiers are also said to be "maximal matching." while non-greedy are also called "lazy." "ungreedy," and "minimal matching," among others.* Table 7-4 summarizes:

Table 7-4: Perl's Quantifiers (Greedy and Lazy)

Number of matches	**Traditional Greedy** (maximal matching)	**Lazy**☆ **(Non-greedy)** (minimal matching)
Any number (zero, one, or more)	`*`	`*?`
One or more	`+`	`+?`
Optional (zero or one)	`?`	`? ?`
Specified limits (at least *min*: no more than *max*)	`{min, max}`	`{min, max}?`
Lower limit (at least *min*)	`{min,}`	`{min,} ?`
Exactly *num*†	`{num}`	`{num}?`

☆ Not available before Perl5.

† There is nothing optional to be matched by an "exactly *num*" construct, so both versions are identical except for efficiency they co-exist only as a matter of notational convenience and consistency.

The non-greedy versions are examples of the "ungainly looking" additions to the regex flavor that appeared with Perl5. Traditionally, something like `*?` makes no sense in a regex. In fact, in Perl4 it is a syntax error. So, Larry was free to give it new meaning. There was some thought that the non-greedy versions might be `**`, `++`, and such, which has certain appeal, but the problematic notation for 「`{min,max}`」 led Larry to choose the appended question mark. This also leaves 「`**`」 and the like for future expansion.

---

* Larry Wall's preferred perlance uses *minimal matching* and *maximal matching.*

Non-greedy efficiency

Many of the effects that non-greedy quantifiers have on efficiency were discussed in Chapter 5's "A Detailed Look at the Performance Effects of Parentheses and Backtracking," starting on page 151, and "Simple Repetition" on page 155. Other effects stem primarily from the regex-directed nature of Perl's NFA engine.

It's not common that you have a free choice between using greedy and non-greedy quantifiers, since they have such different meanings, but if you do, the choice is highly dependent upon the situation. Thinking about the backtracking that either must do with the data you expect should lead you to a choice. For an example, I recommend my article in the Autumn 1996 (Volume 1, Issue 3) issue of *The Perl Journal* in which I take one simple problem and investigate a variety of solutions, including those with greedy and non-greedy quantifiers.

A non-greedy construct vs. a negated character class

I find that people often use a non-greedy construct as an easily typed replacement for a negated character-class, such as using ⌈`<(.+?)>`⌋ instead of ⌈`<([^>]+)>`⌋. Sometimes this type of replacement works, although it is less efficient the implied loop of the star or plus must keep stopping to see whether the rest of the regex can match. Particularly in this example, it involves temporarily leaving the parentheses which, as Chapter 5 points out, has its own performance penalty (☞ 150). Even though the non-greedy constructs are easier to type and perhaps easier to read, make no mistake: *what they match can be very different.*

First of all, of course, is that if the `/s` modifier (☞ 234) is not used, the dot of ⌈`.+?`⌋ doesn't match a newline, while the negated class in ⌈`[^>]+`⌋ does.

A bigger problem that doesn't always manifest itself clearly can be shown with a simple text markup system that uses <…> to indicate emphasis, as with:

```
Fred was very, <very> angry. <Angry!> I tell you.
```

Let's say you want to specially process marked items that end with an exclamation point (perhaps to add extra emphasis). One regex to match those occurrences and capture the marked text is ⌜`<([^>]*!)>`⌟. Using ⌜`<(.*?!)>`⌟, even with the /s modifier, is very different. The former matches '…`Angry!`…' while the latter matches …`very> angry. <Angry!`…'.

The point to remember is that the negated class in ⌜`[^>]*>`⌟ *never* matches '`>`', while the non-greedy construct in ⌜`.*?>`⌟ does *if that is what it takes to achieve a match.* If nothing after the lazy construct could force the regex engine to backtrack, it's not an issue. However, as the exclamation point of this example illustrates, the desire for a match allows the lazy construct past a point you really don't want it to (and that a negated character can't) exceed.

The non-greedy constructs are without a doubt the most powerful Perl5 additions to the regex flavor, but you must use them with care. A non-greedy ⌈`.*?`⌋ is almost never a reasonable substitute for ⌈`[^…]*`⌋—one might be proper for a particular situation, but due to their vastly different meaning, the other is likely incorrect.

Grouping

As I've noted often, parentheses traditionally have had two uses: for grouping and for capturing matched text to `$1`, `$2`, and friends. Within the same regular expression, the metacharacters `\1`, `\2`, and so on, are used instead. As pointed out on page 219, the difference is much more than just notational. Except within a character class where a backreference makes no sense, `\1` through `\9` are always backreferences. Additional backreferences (`\10`, `\11`, …) become available as the number of capturing parentheses warrants (☞ 243).

Uniquely, Perl provides two kinds of parentheses: the traditional `(…)` for both grouping and capturing, and new with Perl5, the admittedly unsightly `(?:…)` for grouping only. With `(?:…)`, the "opening parenthesis" is really the three-character sequence '`(?:`', while the "closing parenthesis" is the usual ')'.

Like the notation for the non-greedy quantifiers, the sequence '`(?`' was previously a syntax error. Starting with version 5, it is used for a number of regex language extensions, of which `(?:…)` is but one. We'll meet the others soon.

Capturing vs. non-capturing parentheses

The benefits in a grouping-only, non-capturing construct include:

- Efficient matching from bypassing the (perhaps non-trivial) overhead involved with capturing text you don't intend to use.

- The possibility of greater internal optimizations because subexpressions that don't need to be isolated for capturing purposes might be converted to something more efficient by the regex engine. ⌈`(?:return-to|reply-to):·`⌋ and ⌈`re(?:turn-to:·|ply-to:·)`⌋, for example, are logically equivalent, but the latter is faster, with both matches and failures. (If you're not convinced, work through an attempt to match against **`forward-·and·reply-to·fields`**…) Perl's regex engine does not currently take advantage of this, but it could in the future.

- Ease in building a usable regular expression in a string.

This last item is probably the one that provides the most benefit from a user's point of view: Recall in the CSV example how because the first alternative used two sets of parentheses, those in the second alternative captured to `$3` (☞ 204; ☞ 207). Any time the first alternative changes the number of parentheses it uses, all subsequent references must be changed accordingly. It's a real maintenance nightmare that non-capturing parentheses can greatly alleviate.*

For related reasons, the number of capturing parentheses becomes important when `m/…/` is used in a list context, or at any time with `split`. We will see specific examples in each of the relevant sections later (☞ 252, 264).

Lookahead

Perl5 also added the special `(?=…)` and `(?!…)` lookahead constructs. Like normal non-capturing parentheses, the positive lookahead ⌈`(?=`*subexpression*`)`⌋ is true if the subexpression matches. The twist is that the subexpression does not actually "consume" any of the target string lookahead matches a *position* in the string similar to the way a word boundary does. This means that it doesn't add to what `$&` or any enclosing capturing parentheses contain. It's a way of peeking ahead without taking responsibility for it.

Similarly, the negative lookahead ⌈`(?!`*subexpression*`)`⌋ is true when the subexpression does not match. *Superficially*, negative lookahead is the logical counterpart to a negated character class, but there are two major differences:

- A negated character class must match *something* to be successful, and consumes that text during the match. Negative lookahead is successful when it can't match anything. The second and third examples below illustrate this.
- A character class (negated or not) matches a single character in the target text. Lookahead (positive or negative) can test an arbitrarily complex regular expression.

A few examples of lookahead:

⌈`Bill(?=·The·Cat|·Clinton)`⌋

Matches `Bill`, but only if followed by '`·The·Cat`' or '`·Clinton`'.

⌈`\d+(?!\.)`⌋

Matches a number if not followed by a period.

⌈`\d+(?=[^.])`⌋

Matches a number if followed by something other than a period. Make sure you understand the difference between this and the previous item   consider a number at the very end of a string. Actually, I'll just put that question to you now. Which of these two will match the string '`OH•44272`', and where? ❖ Think carefully, then turn the page to check your answer.

---

* Even better still would be named subexpressions, such as Python provides through its *symbolic group names,* but Perl doesn't offer this . . . yet.

⌜`^(?![A-Z]*$)[a-zA-Z]*$`⌟

Matches if the item contains only letters, but not all uppercase.

⌜`^(?=.*?this)(?=.*?that)`⌟

A rather ingenious, if not somewhat silly, way of checking whether ⌜`this`⌟ *and* ⌜`that`⌟ can match on the line. (A more logical solution that is mostly comparable is the dual-regex `/this/ && /that/`.)*

Other illustrative examples can be found at "Keeping the Match in Synch with Expectations" (☞ 237). For your entertainment, here's a particularly heady example copied from "Adding Commas to a Number" (☞ 292):

```
s<
    (\d{1,3})          #  before a comma: one to three digits
    (?=                #  followed by, but not part of what's matched. . .
       (?:\d\d\d)+     #       some number of triplets. . .
       (?!\d)          #     . . . not followed by another digit
    )                  #     (in other words, which ends the number)
>< $1, >gx;
```

Lookahead parentheses do not capture text, so they do not count as a set of parentheses for numbering purposes. However, they may *contain* raw parentheses to capture the phantomly matched text. Although I don't recommend this often, it can perhaps be useful. For example, ⌜`(.*?)(?=<(strong|em)\s*>)`⌟, matches everything up to *but not including* a `<strong>` or `<em>` HTML tag on the line. The consumed text is put into `$1` (and `$&` as well, of course), while the `'strong'` or `'em'` that allows the match to stop is placed into `$2`. If you don't need to know which tag allows the match to stop successfully, the ⌜`…(strong|em)…`⌟ is better written as ⌜`…(?:strong|em)…`⌟ to eliminate the needless capturing. As another example, I use an appended ⌜`(?=(.*))`⌟ on page ☞ 277 as part of mimicking `$'`. (Due to the $&-penalty, we want to avoid `$'` if at all possible; ☞ 273.)

Using capturing parentheses within a negative lookahead construct makes absolutely no sense, since the lookahead construct matches only when its enclosed subexpression does not.

The `perlre` manpage rightly cautions that look*ahead* is very different from look*behind.* Lookahead ensures that the condition (matching or not matching the given subexpression) is true starting at the current location and looking, as normal, toward the right. Lookbehind, were it supported, would somehow look back toward the left.

---

* There are quite a variety of ways to implement "⌈`this`⌋ *and* ⌈`that`⌋, and as many ways to compare them. Major issues of such a comparison include understanding exactly what is matched, match efficiency, and understandability. I have an article in the Autumn 1996 (Volume 1, Issue 3) issue of *The Perl Journal* that examines this in great detail, providing, I hope, additional insight about what the regex engine's actions mean in the real world. By the way, the ingenious solution shown here comes from a post by Randal Schwartz, and it fared rather well in the benchmarks.

Positive lookahead vs. negative lookahead

❖ Answer to the question on page 228

Both ⌈`\d+(?!\.)`⌋ and ⌈`\d+(?=[^.])`⌋ can match '`OH•44272`'. The first matches `OH•44272`, while the second `OH•44272`.

Remember, greediness always defers in favor of an overall match. Since ⌈`\d+(?=[^.])`⌋ requires a non-period after the matched number, it will give up part of the number to become the non-period if need be.

We don't know what these might be used for, but they should probably be written as ⌈`\d+(?![\d.])`⌋ and ⌈`\d+(?=[^.\d])`⌋.

For example, ⌈`(?!000)\d\d\d`⌋ means "so long as they're not `000`, match three digits," which is a reasonable thing to want to do. However, it is important to realize that it specifically does *not* mean "match three digits that are not preceded by `000`." This would be lookbehind, which is not supported in Perl or any other regex flavor that I know of. Well, actually, I suppose that a leading anchor (either type, string or word) can be considered a limited form of lookbehind.

You often use lookahead at the end of an expression to disallow it from matching when followed (or not followed) by certain things. Although its use at the beginning of an expression might well indicate a mistaken attempt at lookbehind, leading lookahead can sometimes be used to make a general expression more specific. The `000` is one example, and the use of ⌈`(?!0+\.0+\.0+\.0+\b)`⌋ to disallow null IP addresses in Chapter 4 (☞ 125) is another. And as we saw in Chapter 5's "A Global View of Backtracking" (☞ 149), lookahead at the beginning of an expression can be an effective way to speed up a match.

Otherwise, be careful with leading negative lookahead. The expression ⌈`\w+`⌋ happily matches the first word in the string, but prepending ⌈`(?!cat)`⌋ is not enough to ensure "the first word not beginning with cat." ⌈`(?!cat)\w+`⌋ can't match at the start of `cattle`, but can still match `cattle`. To get the desired effect, you need additional precautions, such as ⌈`\b(?!cat)\w+`⌋.

Comments within a regular expression

The ⌈`(?#...)`⌋ construct is taken as a comment and ignored. Its content is not entirely free-form, as any copy of the regex-operand delimiter must still be escaped.[*]

---

[*] `(?#...)` comments are removed very early in the parsing, effectively between Phase A and Phase B of page 223's Figure 7-1. A bit of trivia for you: As far as I can tell, the closing parenthesis of a `(?#...)` comment is the only item in Perl's regular expression language that cannot be escaped. The first closing parentheses after the '`(?#`' ends the comment, period.

`(?#...)` appeared in version 5.000, but as of 5.002, the `/x` modifier enables the unadorned ⌈`#`⌋ comment that runs to the next newline (or the end of the regex), pretty much just like in regular code. (We saw an example of this earlier in the commafication snippet on page 229.) The `/x` modifier also causes most whitespace to be ignored, so you can write the

```
    $text =~ m/"([^"\\]*(\\.[^"\\]*)*)",?|([^,]+),?|,/g
```

from the CSV example as the more-easily understandable:

```
    $text =~ m/                 # A field is one of three types of things:
                                #   1) DOUBLEQUOTED STRING
      "([^"\\]*(\\.[^"\\]*)*)"  #    - Standard doublequoted string (nab to $1).
      ,?                        #    - Eat any trailing comma.
            |                   # -OR-
                                #   2) NORMAL FIELD
      ([^,]+)                   #    - Capture (to $3) up to next comma.
      ,?                        #    - (and including comma if there is one)
            |                   # -OR-
                                #   3) EMPTY FIELD
      ,                         #    just match the comma.
    /gx;
```

The underlying regular expression is exactly the same. As with `(?#...)` comments, the regex's closing delimiter still counts, so these comments are not entirely free-form either. As with most other regex metacharacters, the ⌈`#`⌋ and whitespace metacharacters that become active with `/x` are not available within a character class, so there can be no comments or ignored whitespace within classes. Like other regex metacharacters, you can escape # and whitespace within an `/x` regex:

```
m{
  ^               #  Start of line.
  (?:             #  Followed by one of
     From         #        'From'
     |Subject     #        'Subject'
     |Date        #        'Date'
  )               #
  :               #  All followed by a colon. . .
  \ *             #  .. and any number of spaces. (note escaped space)
  (.*)            #  Capture rest of line (except newline) to $1.
}x;
```

Other `(?...)` constructs

The special ⌜`(?`*modifiers*`)`⌟ notation uses the '`(?`' notation mentioned in this section, but for a different purpose. Traditionally, case-insensitive matching is invoked with the `/i` modifier. You can accomplish the same thing by putting ⌜`(?i)`⌟ anywhere in the regex (usually at the beginning). You can specify `/i` (case insensitive), `/m` (multi-line mode), `/s` (single-line mode), and `/x` (free formatting) using this mechanism. They can be combined; ⌜`(?si)`⌟ is the same as using both `/i` and `/s`.

String Anchors

Anchors are indispensable for creating bullet-proof expressions, and Perl provides several flavors.

Logical lines vs. raw text

Perl provides the traditional string anchors, caret and dollar,[*] but their meaning is a bit more complex than simply "start- and end-of-string." With the typically simple use of a regular expression within a **`while (<>)`** loop (and the *input record separator* `$/` at its default), you know that the text being checked is exactly one logical line, so the distinction between "start of logical line" and "start of the string" is irrelevant.

However, if a string contains embedded newlines, it's reasonable to consider the one string to be a collection of multiple logical lines. There are many ways to create strings with embedded newlines (such as `"this\nway"`) when applying a regex to such a string, however it might have acquired its data, should ⌈`^Subject:`⌋ find '`Subject:`' at the start of any of the logical lines, or only at the start of the entire multi-line string?

Perl allows you to do either. Actually, there are four distinct modes, summarized in Table 7-5:

Table 7- 5: Overview of Newline-Related Match Modes

mode	**^ and $ anchors consider target text as**	**dot**
default mode	a single string, without regard to newlines	doesn't match newline
single-line mode	a single string, without regard to newlines	*matches all characters*
multi-line mode	*multiple logical lines separated by newlines*	(unchanged from default)
clean multi-line	*multiple logical lines separated by newlines*	*matches all characters*

When there are no embedded newlines in the target string, all modes are equal.[**] You'll noticed that Table 7-5 doesn't mention. . .

- **how** the target text might acquire multiple logical lines, that is, embedded newlines (*it's irrelevant*);
- **when** you may apply a match to such a string (*anytime you like*); or
- **whether** ⌈`\n`⌋, an appropriate octal escape, or even ⌈`[^x]`⌋ can match a newline (*they aluays can*).

---

[*] A dollar sign can also indicate variable interpolation, but whether it represents that or the end-of-line metacharacter is rarely ambiguous. Details are in "Doublequotish Processing and Variable Interpolation" (☞ 222).

[**] Well, to be pedantic, the line anchor optimization becomes slightly less effective in the two multi-line modes, since the transmission must apply the regex after any embedded newline (☞ 158).

In fact, meticulous study of Table 7-5 reveals that *these modes are concerned only with how the three metacharacters, caret, dollar, and dot, consider newlines.*

The default behavior of caret, dollar, and dot

Perl's default is that caret can match only at the beginning of the string. Dollar can match at the end of the string, *or* just before a newline at the end of the string. This last provision might seem a bit strange, but it makes sense in the context how Perl is normally used: input is read line by line, with the trailing newline kept as part of the data. Since dot doesn't match a newline in the default mode, this rule allows ⌈`.*$`⌋ to consume everything up to, but not including, the newline. My term for this default mode is *the default mode.* You can quote me.

`/m` and the ill-named "multi-line mode"

Use the `/m` modifier to invoke a *multi-line mode* match. This allows caret to match at the beginning of any logical line (at the start of the string, as well as after any embedded newline), and dollar to match at the end of any logical line (that is, before any newline, as well as at the end of the string). The use of `/m` does not effect what dot does or doesn't match, so in the typical case where `/m` is used alone, dot retains its default behavior of not matching a newline. (As we'll soon see, `/m` can be combined with `/s` to create a *clean multi-line mode* where dot matches anything at all.)

Let me say again:

> The /m modifier influences *only* how ⌈`^`⌋ and ⌈`$`⌋ treat newlines.

The `/m` modifier affects *only* regular-expression matching, and in particular, *only* the caret and dollar metacharacters. It has *nothing whatsoever* to do with anything else. Perhaps "multi-line mode" is better named "line-anchors-notice-embedded-newlines mode." The `/m` and multi-line mode are debatably the most misunderstood simple features of Perl, so please allow me to get on the soapbox for a moment to make it clear that the `/m` modifier has **nothing** to do with. . .

- Whether you can work with data that has newlines in it. You can *always* work with any type of data. ⌈`\n`⌋ always matches a newline without regard to multiline mode, or the lack thereof.

- Whether dot (or anything else) does or does not match a newline. The `/s` modifier (discussed on the next page) influences dot, but the `/m` modifier does not. The `/m` modifier influences only whether the anchors match *at* a newline. Conceptually, this is not entirely unrelated to whether dot matches *a* newline, so I often bring the description of dot's default behavior under the umbrella term "multi-line mode," but the relationship is circumstantial.

- How data, multi-line or not, finds its way to the target string. Multi-line mode is often used in conjunction with other useful features of Perl, but this relationship, too, is circumstantial. A prime example is `$/`, the *input-record separator* variable. If `$/` is set to the empty string, it puts `<>` and the like into *paragraph mode* in which it will return, as a single string, all the lines until (and including) the next blank line. If you set `$/` to `undef`, Perl goes into *file slurp* mode where the entire (rest of the) file is returned, all in a single string.*

  It's convenient to use multi-line mode in conjunction with a special setting of `$/`, but neither has any relationship to the other.

The `/m` modifier was added in Perl5-Perl4 used the now-obsolete special variable `$*` to indicate multi-line mode for *all* matches. When `$*` is true, caret and dollar behave as if `/m` were specified. This is less powerful than explicitly indicating that you want multi-line mode on a per-use basis, so modern programs should not use `$*`. However, modern programmers *should* worry if some old or unruly library does, so the complement of `/m` is the `/s` modifier.

Single-line mode

The `/s` modifier forces caret and dollar to not consider newlines as special, even if `$*` somehow gets turned on. It also affects dot: with `/s`, dot matches *any* character. The rationale is that if you go to the trouble to use the `/s` modifier to indicate that you are not interested in logical lines, a newline should not get special treatment with dot, either.

Clean multi-line mode

Using the `/m` and `/s` modifiers together creates what I call a "clean" multi-line mode. It's the same as normal multi-line mode except dot can match any character (the influence added by `/s`). I feel that removing the special case of dot not matching newline makes for cleaner, more simple behavior, hence the name.

Explicit start- and end-of-string

To allow greater flexibility, Perl5 also provides ⌈`\A`⌋ and ⌈`\Z`⌋ to match the beginning and end of the string. They are never concerned with embedded newlines. They are exactly the same as the default and `/s` versions of caret and dollar, but they can be used even with the `/m` modifier, or when `$*` is true.

---

* A non-regex warning: It's a common mistake to think that undefining `$/` causes the next `<>` read to return "the rest of the input" (or if used at the start of the script, "all the input"). "File slurp" mode is still a *file* slurp mode. If there are multiple files for `<>` to read, you still need one call per file. If you really want to get *all* the input, use **`join('',<>)`**.

Newline is always special in one respect

There is one situation where newline always gets preferential treatment. Regardless of the mode, both ⌈`$`⌋ and ⌈`\Z`⌋ are always allowed to match before a *text-ending newline.* In Perl4, a regex cannot require the absolute end of the string. In Perl5, you can use ⌈`…(?!\n)$`⌋ as needed. On the other hand, if you want to force a trailing newline, simply use ⌈`…\n$`⌋ in any version of Perl.

Eliminating warnings about `$*`

Perl scripts should generally not use `$*`, but sometimes the same code needs to support Perl4 as well. Perl5 issues a warning if it sees `$*` when warnings are turned on (as they generally should be). The warnings can be quite annoying, but rather than turning them off for the entire script, I recommend:

```
     { local($^W) = 0;  eval '$* = 1' }
```

This turns off warnings while `$*` is modified, yet when done leaves warnings on if they were on—I explained this technique in detail in "Dynamic Scope" (☞ 213).

`/m` vs. `(?m)`, `/s` vs. `/m`

As mentioned briefly on page 231, you can specify certain modifiers using the ⌈`(?mod)`⌋ notation within the regex itself, such as using ⌈`(?m)`⌋ as a substitute for using `/m`. There are no fancy rules regarding how the `/m` modifier might conflict with ⌈`(?m)`⌋, or where ⌈`(?m)`⌋ can appear in the regex. Simply using either `/m` or ⌈`(?m)`⌋ (anywhere at all in the regex) enables multi-line mode for the entire match.

Although you may be tempted to want something like ⌈`(?m)…(?s)…(?m)…`⌋ to change the mode mid-stream, the line mode is an all-or-nothing characteristic for the entire match. It makes no difference how the mode is specified.

Combining both `/s` and `/m` has `/m` taking precedence with respect to caret and dollar. Still, the use or non-use of `/m` has no bearing on whether a dot matches a newline or not   only the explicit use of `/s` changes dot's default behavior. Thus, combining both modes creates the clean multi-line mode.

All these modes and permutations might seem confusing, but Table 7-6 should keep things straight. Basically, they can be summarized with "`/m` means multi-line, `/s` means dot matches newline."

All other constructs are unaffected. ⌈`\n`⌋ always matches a newline. A character class can be used to match or exclude the newline characters at any time. An inverted character class such as ⌈`[^x]`⌋ always matches a newline (unless `\n` is included, of course). Keep this in mind if you want to change something like ⌈`.*`⌋ to the seemingly more restrictive ⌈`[^…]*`⌋.

Table 7-6: Summary of Anchor and Dot Modes

Mode	**Specified With**	`^`	`$`	`\A, \Z`	**Dot Matches Newline**	
default	neither `/s` nor `/m`, `$*` false	string	string	string	no	
single-line	`/s` (`$*` irrelevant)	string	string	string	*yes*	
multi-line	`/m` (`$*` irrelevant)	*line*	*line*	string	no	[unchanged from default]
clean multi-line	both `/m` and `/s` (`s*` irrelevant)	*line*	*line*	string	*yes*	
obsolete multi-line	neither /s nor /m: $* true	*line*	*line*	string	no	[unchanged from default]

string	cannot anchor to an embedded newline.
line	can anchor to an embedded newline.

Multi-Match Anchor

Perl5 adds the ⌈`\G`⌋ anchor, which is related to ⌈`\A`⌋, but is geared for use with `/g`. It matches at the point where the previous match left off. For the first attempt of a `/g` match, or when `/g` is not used, it is the same as ⌈`\A`⌋.

An example with `\G`

Let's look at a lengthy example that might seem a bit contrived, but which illustrates some excellent points. Let's say that your data is a series of five-digit US postal codes (ZIP codes) that are run together, and that you need to retrieve all that begin with, say, `44`. Here is a sample line of data, with the target codes in bold:

`03824531449411615213`**`44182`**`95035`**`44272`**`752010217443235`

As a starting point, consider that we can use **`@zips = m/\d\d\d\d\d/g;`** to create a list with one ZIP code per element (assuming, of course, that the data is in the default search variable `$_`). The regular expression matches one ZIP code each time `/g` iteratively applies it. A point whose importance will soon become apparent: the regex never fails until the entire list has been parsed—there are absolutely no bump-and-retries by the transmission. (I'm assuming we'll have only proper data, an assumption that is sometimes valid in the real world—but usually not.)

So, it should be apparent that changing ⌈`\d\d\d\d\d`⌋ to ⌈`44\d\d\d`⌋ in an attempt to find only ZIP codes starting with `44` is silly—once a match attempt fails, the transmission bumps along one character, thus putting the match for the ⌈`44`⌋ out of synch with the start of each ZIP code. Using ⌈`44\d\d\d`⌋ incorrectly finds …`531`**`44941`**`16`… as the first match.

You could, of course, put a caret or ⌈`\A`⌋ at the head of the regex, but they allow a target ZIP code to match only if it's the first in the string. We need to keep the regex engine in synch manually by writing our regex to pass over undesired ZIP codes as needed. The key here is that it must pass over full ZIP codes, not single characters as with the automatic bump-along.

Keeping the match in synch with expectations

I can think of several ways to have the regex pass over undesired ZIP codes. Any of the following inserted at the head of the regex achieves the desired effect:

⌈`(?:[^4]\d\d\d\d|\d[^4]\d\d\d)*...`⌋

> This brute-force method actively skips ZIP codes that start with something other than `44`. (Well, it's probably better to use ⌈`[1235-9]`⌋ instead of ⌈`^4`⌋, but as I said earlier, I am assuming properly formated data.) By the way, we can't use ⌈`(?:[^4][^4]\d\d\d)*`⌋, as it does not pass over undesired ZIP codes like `43210`.

⌈`(?:(?!44)\d\d\d\d\d)*...`⌋

> This similar method actively skips ZIP codes that do not start with `44`. This English description sounds virtually identical to the one above, but when rendered into a regular expression looks quite different. Compare the two descriptions and related expressions. In this case, a desired ZIP code (beginning with `44`) causes ⌈`(?!44)`⌋ to fail, thus causing the skipping to stop.

⌈`(?:\d\d\d\d\d)*?...`⌋

> This method skips ZIP codes only when needed (that is, when a later subexpression describing what we *do* want fails). Because of the minimal-matching, ⌈`(?:\d\d\d\d\d)`⌋ is not even attempted until whatever follows has failed (and is repeatedly attempted until whatever follows finally does match, thus effectively skipping only what is absolutely needed).

Combining this last method with ⌈`(44\d\d\d)`⌋ gives us

```
@zips = m/(?:\d\d\d\d\d)*?(44\d\d\d)/g;
```

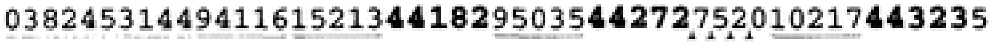
and picks out the desired '`44xxx`' codes, actively skipping undesired ones that intervene. (When used in a list context, `m/.../g` returns a list of the text matched by subexpressions within capturing parentheses from each match; ☞ 253.)

This regex can work with `/g` because we know each match always leaves the "current match position" at the start of the next ZIP code, thereby priming the next match (via `/g`) to start at the beginning of a ZIP code as the regex expects. You might remember that we used this same keeping-in-synch technique with the CSV example (☞ 206).

Hopefully, these techniques are enlightening, but we still haven't seen the ⌈`\G`⌋ we're supposed to be looking at. Continuing the analysis of the problem, we find a use for ⌈`\G`⌋ quickly.

Maintaining synch after a non-match as well

Have we *really* ensured that the regex will always be applied only at the start of a ZIP code? *No!* We have manually skipped *intervening* undesired ZIP codes, but once there are no more desired ones, the regex will finally fail. As always, the bump-along-and-retry happens, thereby starting the match from a position *within* a ZIP code! This is a common concern that we've seen before, in Chapter 5's Tcl program to remove C comments (☞ 173). Let's look at our sample data again:



0382453144941161521 3**44182**95035**44272**752010217**44323**5

Here, the matched codes are bold (the third of which is undesired), the codes we actively skipped are underlined, and characters skipped via bump-along-and-retry are marked. After the match of `44272`, no more target codes are able to be matched, so the subsequent attempt fails. Does the whole match attempt end? Of course not. The transmission bumps along to apply the regex at the next character, putting us out of synch with the real ZIP codes. After the fourth such bump-along, the regex skips `10217` as it matches the "ZIP code" `44323`.

Our regex works smoothly so long it's applied at the start of a ZIP code, but the transmission's bump-along defeats it. This is where ⌈`\G`⌋ comes in:

```
@zips = m/\G(?:\d\d\d\d\d)*?(44\d\d\d)/g;
```

`\G` matches the point where the previous `/g` match ended (or the start of the string during the first attempt). Because we crafted the regex to explicitly end on a ZIP code boundary, we're assured that any subsequent match beginning with `\G` will start on that same ZIP code boundary. If the subsequent match fails, we're done for good because a bump-along match is impossible—`\G` requires that we start from where we had last left off. In other words, this use of `\G` effectively disables the bump along.

In fact, the transmission is optimized to actually disable the bump-along in common situations. If a match must start with `\G`, a bump-along can never yield a match, so it is done away with altogether. The optimizer can be tricked, so you need to be careful. For example, this optimization isn't activated with something like ⌈`\Gthis|\Gthat`⌋, even though it is effectively the same as ⌈`\G`**`(?:`**`this`**`|`**`that`**`)`**⌋ (which it *does* optimize).

\G in perspective

⌈`\G`⌋ is not used often, but when needed, it is indispensable. As enlightening as I hope this example has been, I can actually see a way to solve it without `\G`. In the interest of study, I'd like to mention that after successfully matching a `44`*`xxx`* ZIP code, we can use either of the first two "skip undesired ZIP codes" subexpressions to bypass any *trailing* undesired ZIP codes as well (the third bypasses only when forced, so would not be appropriate here):

```
@zips = m/(?:\d\d\d\d\d)*?(44\d\d\d)(?:(?!44)\d\d\d\d\d)*/g;
```

After the last desired ZIP code has been matched, the added subexpression consumes the rest of the string, if any, and the `m/…/g` is finished.

These methods work, but frankly, it is often prudent to take some of the work out of the regular expression using other regular expressions or other language features. The following two examples are easier to understand and maintain:

```
@zips = grep {defined} m/(44\d\d\d)|\d\d\d\d\d/g;
@zips = grep {m/^44/} m/\d\d\d\d\d/g;
```

In Perl4, you *can't* do it all in one regex because it doesn't have many of the constructs we've used, so you need to use a different approach regardless.

Priming a `/g` match

Even if you don't use `\G`, the way Perl remembers the "end of the previous match" is a concern. In Perl4, it is associated with a particular regex operator, but in Perl5 it is associated with the matched data (the target string) itself. In fact, this position can be accessed using the `pos(…)` function. This means that one regex can actually pick up where a different one left off, in effect allowing multiple regular expressions to do a tag-team match. As a simple example, consider using

```
@nums = $data =~ m/\d+/g;
```

to pick apart a string, returning in `@nums` a list of all numbers in the data. Now, let's suppose that if the special value `<xx>` appears on the line, you want only numbers after it. An easy way to do it is:

```
     $data =~ m/<xx>/g;    # prime the /g start. pos($data) now points tojust after
the <xx>.
     @nums = $data =~ m/\d+/g;
```

The match of `<xx>` is in a scalar context, so `/g` doesn't perform multiple matches (☞ 253). Rather, it sets the `pos` of `$data`, the "end of the last match" position where the next `/g`-governed match of the same data will start. I call this technique *priming* the `/g`. Once done, `m/\d+/g` picks up the match at the primed point. If ⌈`<xx>`⌋ can't match in the first place, the subsequent `m/\d+/g` starts at the beginning of the string, as usual.

Two important points allow this example work. First, the first match is in a scalar context. In a list context, the ⌜`<xx>`⌟ is applied repeatedly until it fails, and failure of a `/g`-governed match resets `pos` to the start of the string. Secondly, the first match must use `/g`. Matches that don't use `/g` never access `pos`.

And here is something interesting: Because you can assign to `pos`, you can prime the `/g` start manually:

```
pos($data) = $i if $i = index($data,"<xx>"), $i > 0;
@nums = $data =~ m/\d+/g;
```

If `index` finds `<xx>` in the string, it sets the start of the next `/g-`governed match of `$data` to begin there. This is slightly different from the previous example - here we prime it to start at the `<xx>`, not after it as we did before. It turns out, though, that it doesn't matter in this case.

This example is simple, but you can imagine how these techniques could be quite useful if used carefully in limited cases. You can also imagine them creating a maintenance nightmare if used carelessly.

Word Anchors

Perl lacks separate *start-of-word* and *end-of-word* anchors that are commonly found in other tools (Table 3-1 ☞ 63; Table 6-1 ☞ 182). Instead,* Perl has *word-boundary* and *non-wordboundary* anchors ⌈`\b`⌋ and ⌈`\B`⌋. (Note: in character classes, and in doublequoted strings for that matter, `\b` is a shorthand for a backspace.) A word boundary is any position where the character on one side matches ⌈`\w`⌋ and the character on the other matches ⌈`\W`⌋ (with the ends of the string being considered ⌈`\W`⌋ for the purposes of this definition). Note that unlike most tools, Perl includes the underscore in ⌈`\w`⌋. It can also include additional characters if a locale is defined in the user environment (☞ 65, 242).

An important warning about ⌈`\b`⌋

There's one particular danger surrounding ⌈`\b`⌋ that I sometimes run into.** As part of a Web search interface, given an `$item` to find, I was using `m/\b\Q$item\E\b/` to do the search. I wrapped the item in `\b`…`\b` because I knew I wanted to find only whole-word matches of whatever `$item` was. Given an item such as '`3.75`', the search regex would become ⌈`\b3\.75\b`⌋, finding things like '`price is 3.75 plus tax`' as expected.

However, if the `$item` were '`$3.75`', the regex would become ⌈`\b\$3\.75\b`⌋, which requires a word boundary before the dollar sign. (The only way a word boundary can fall *before* a `\W` character like a dollar sign is if a word *ends* there.) I'd prepended `\b` with the thought that it would force the match to start where `$item` started its own word. But now that the start of `$item` *can't* start its own word, ('`$`' isn't matched by `\w`, so cannot possibly start a word) the `\b` is an impediment. The regex doesn't even match '…`is $3.75 plus`…'.

We don't even want to begin the match if the character before can match `\w`, but this is lookbehind, something Perl doesn't have. One way to address this issue is to add `\b` only when the `$item` starts or ends with something matching `\w`:

```
    $regex = "\Q$item\E";                          # make $item "safe"
    $regex = '\b' . $regex if $regex =~ m/^\w/; # if can start word, ensure it does
    $regex = $regex . '\b' if $regex =~ m/\w$/; # if can end word, ensure it does
```

This ensures that ⌈`\b`⌋, doesn't go where it will cause problems, but it still doesn't address the situations where it does (that is, where the `$item` begins or ends with

---

* The two styles are not mutually exclusive. GNU Emacs, for example, provides both.

** Most recently, actually, just before feeling the need to write this section!

a `\W` character). For example, an `$item` of `-998` still matches in '`800-998-9938`'. If we don't mind matching more text than we really want (something that's not an option when embedded within a larger subexpression, and often not an option when applied using the `/g` modifier), we can use the simple, but effective, ⌈`(?:\W|^)\Q$item\E(?!\w)`⌋.

Dealing with a lack of separate start and end anchors

At first glance, Perl's lack of separate start- and end-of-word anchors might seem to be a major shortcoming, but it's not so bad since ⌈`\b`⌋'s use in the regex almost always disambiguates it. I've never seen a situation where a specific start-only or end-only anchor was actually needed, but if you ever do bump into one, you can use ⌈`\b(?=\w)`⌋ and ⌈`\b(?!\w)`⌋ to mimic them. For example,

```
s/\b(?!\w).*\b(?=\w)//
```

removes everything between the first and last word in the string. As a comparison, in modern versions of *sed* that support the separate `\<` and `\>` word boundaries, the same command would be `s/\>.*\<//`.

Convenient Shorthands and Other Notations

We've already seen many of Perl's convenient shorthands for common constructs. Table 7-7 shows the full* list.

Table 7-7: Regex Shorthands and Special-Character Encodings

	Byte Notations		**Machine-dependent**
`\` *num*	character specified in octal		**Control Characters**
`\x` *num*	character specified in hexadecimal	`\a`	alarm (bell)
`\c` *char*	control character	`\f`	formfeed
	Shorthand for Common Classes	`\e`	escape
`\d`	digit `[0-9]`	`\n`	newline
`\s`	whitespace, usually `[·\f\n\r\t]`	`\r`	carriage return
`\w`	word character, usually `[a-zA-Z0-9_]`	`\t`	tab
`\D, \S, \w`	complement of `\d`, `\s`, `\w`	`\b`	backspace
			(only within a class)

The ⌈`\n`⌋ and other shorthands probably seem familiar these machine dependent (☞ 72) notations for common control characters are also available in doublequoted strings. I feel it is important to maintain the mental distinction that these regular expression metacharacters themselves are not available within strings, but that strings just happen to have their own metacharacters which parallel the regex ones in this area (☞ 41).

---

* The phantom ⌈`\v`⌋ (vertical tab) has been omitted. The manpage and other documentation listed it for years, but it was never actually added to the language! I doubt it will be missed now that it has finally been removed from the documentation.

If you compare `m/(?:\r\n)+$/` with

```
$regex = "(\r\n)+";
m/$regex$/;
```

you'll find that they produce exactly the same results. But make no mistake, they are not the same. Naively use something like "`\b[+\055/*]\d+\b`" in the same situation and you'll find a number of surprises. When you assign to a string, it's a *string*—your intention to use it as a regex is irrelevant to how the string processes it. The two `\b` in this example are *intended* to be word boundaries, but to a doublequoted string they're shorthands for a backspace. The regex will never see ⌈`\b`⌋, but instead raw backspaces (which are unspecial to a regex and simply match backspaces). Surprise!

On a different front, both a regex and a doublequoted string convert `\055` to a dash `(055` is the ASCII code for a dash), but if the regex does the conversion, it doesn't see the result as a metacharacter. The string doesn't either, but the resulting `[+-/*]` that the regex eventually receives has a dash that will be interpreted as part of a class range. Surprise!

Finally, `\d` is not a known metasequence to a doublequoted string, so it simply removes the backslash. The regex sees ⌈`d`⌋. Surprise!

I want to emphasize all this because you *must* know what are and aren't metacharacters (and whose metacharacters they are, and when, and in what order they're processed) when building a string that you later intend to use as a regex. It can certainly be confusing at first. An extended example is presented in "Matching an Email Address" (☞ 294).

Perl and the POSIX locale

As "Backdoor Support for Locales" (☞ 65) briefly noted, Perl's support for POSIX locales is very limited. It knows nothing about collating elements and the like, so ASCII-based character-class ranges don't include any of the locale's non-ASCII characters. (String comparisons aren't locale-based either, so neither is sorting.)

If compiled with appropriate libraries, though, Perl uses the "is this a letter?" routines (`isalpha`, `isupper`, and so on), as well as the mappings between uppercase and lowercase. This affects `/i` and the items mentioned in Table 7-8 (☞ 245). It also allows `\w`, `\W`, `\s`, `\S`, (but not `\d`), and word boundaries to respond to the locale. (Perl's regex flavor, however, explicitly does not support `[:digit:]` and the other POSIX bracket expression character classes listed on page 80.)

Related standard modules

Among Perl's standard modules are the `POSIX` module and Jarkko Hietanierni's `I18N::Collate` module. (*I18n* is the common abbreviation for *internationalization*—why is an exercise for your free time.) Although they don't provide regular-expression support, you might find them useful if locales are a concern. The `POSIX` module is huge, but the documentation is relatively brief for additional documentation, try the corresponding C library manpages, or perhaps Donald Lewine's *POSIX Programmer's Guide* (published by O'Reilly & Associates).

Byte notations

Perl provides methods to easily insert bytes using their raw values. Two- or three-digit octal values may be given like ⌈`\33`⌋ and ⌈`\177`⌋, and one- or two-digit hexadecimal values like ⌈`\xA`⌋ and ⌈`\xFF`⌋.

Perl strings also allow one-digit octal escapes, but Perl regexes generally don't because something like ⌈`\1`⌋ is usually taken as a backreference. In fact, multiple-digit backreferences are possible if there are enough capturing parentheses. Thus, ⌈`\12`⌋ is a backreference if the expression has at least 12 sets of capturing parentheses, an octal escape (for decimal 10) otherwise. Upon reading a draft of this chapter, Wayne Berke offered a suggestion that I wholeheartedly agree with: never use a two-digit octal escape such as `\12`, but rather the full three-digit `\012`. Why? Perl will never interpret `\012` as a backreference, while `\12` is in danger of suddenly becoming a backreference if the number of capturing parentheses warrants.

There are two special cases: A backreference within a character class makes no sense so single-digit octal escapes are just peachy within character classes (which is why I wrote *generally don't* in the previous paragraph). Secondly, ⌈`\0`⌋ is an octa*l* escape everywhere, since it makes no sense as a backreference.

Bytes vs. characters, newline vs. linefeed

As Chapter l's "Regex Nomenclature" (☞ 26) explained, exactly which *characters* these *bytes* represent is context-dependent. Usually the context is ASCII, but it can change with the whim of the user or the data. A related concern is that the value imparted by `\n` and friends is not defined by Perl, but is system-dependent (☞ 72).

Character Classes

Perl's class sublanguage is unique among regex flavors because it fully supports backslash escapes. For instance, ⌜`[\-\],]`⌟ is a single class to match a dash, right bracket, and comma. (It might take a bit for ⌜**`[\-\],]`**⌟ to sink in—parse it carefully and it should make sense.) Many other regex flavors do not support backslashes within classes, which is too bad because being able to escape class metacharacters is not only logical, but beneficial. Furthermore, it's great to be allowed to escape items even when not strictly necessary, as it can enhance readability.[*]

---

[*] I generally use GNU Emacs when writing Perl code, and use `cperl-mode` to provide automatic indenting and smart colorization. I often escape quotes within a regex because otherwise they confuse `cperl-mode`, which doesn't understand that quotes within a regex don't start strings.

As I've cautioned throughout this book, metacharacters recognized within a character class are distinct from those recognized outside. Perl is no exception to this rule, although many metacharacters have the same meaning in both situations. In fact, the dual-personality of `\b` aside, everything in Table 7-7 is also supported within character classes, and I find this extremely convenient.

Many normal regex metacharacters, however, are either unspecial or utterly different within character classes. Things like star, plus, parentheses, dot, alternation, anchors and the like are all meaningless within a character class. We've seen that `\b`, `\3`, and `^` have special meanings within a class, but they are unrelated to their meaning outside of a class. Both `-` and `]` are unspecial outside of a class, but special inside (usually).

Character classes and non-ASCII data

Octal and hexadecimal escapes can be quite convenient in a character class, particularly with ranges. For example, the traditional class to match a "viewable" ASCII character (that is, not whitespace or a control character) has been ⌈`[!-~]`⌋. (An exclamation point is the first such character in ASCII, a tilde the last.) It might be less cryptic to spell it out exactly: ⌈`[\x21-\x7e]`⌋. Someone who already knew what you were attempting would understand either method, but someone happening upon ⌈`[!-~]`⌋ for the first time would be confused, to say the least. Using ⌈`[\x21-\x7e]`⌋ at least offers a clue that it is a character-encoding range.

Lacking real POSIX locale support, octal and hexadecimal escapes are quite useful for working with non-ASCII text. For instance, when working with the *Latin-1* (`ISO-8859-1`) encoding popular on the Web, you need to consider that `u` might also appear as `ú`, `ú`, `û`, or `ü` (using the character encodings `\xf9` through `\xfc`). Thus, to match any of these u's, you can use ⌈`[u\xf9-\xfc]`⌋. The uppercase versions are encoded from `\xd9` through `\xdc`, so a case-insensitive match is ⌈`[uU\xf9-\xfc \xd9-\xdc]`⌋ (Using the `/i` modifier applies only to the ASCII `u` so it is just as easy to include `U` directly and save ourselves the woe of the `/i` penalty; ☞ 278.)

Sorting is a practical use of this technique. Normally, to sort `@items`, you simply use **`sort @items`**, but this sorts based on raw byte values and puts `u` (with ASCII value `\x75`) far away from `û` and the like. If we make a copy of each item to use as a sort key (conveniently associating them with an associated array), we can modify the key so that it will work directly with `sort` and yield the results we want. We can then map back to the unmodified key while keeping the sorted order.

Here's a simple implementation:

```
foreach $item (@Items)  {
  $key = lc $item;                      #  Copy $item, forcing ASCII to lowercase.
  $key =~ s/[\xd9-\xdc\xf9-\xfc]/u/g;  #  All types of accented u become plain u
 . . . same treatment for other accented letters. . .
  $pair{$item} = $key;                  #  Remember the item->sortkey relation
}
#  Sort the items based upon their key.
@SortedItems = sort { $pair{$a} cmp $pair{$b} } @Items;
```

(`lc` is a convenient Perl5 feature, but the example is easily rewritten in Perl4.) In reality, this is only the beginning of a solution since each language has its own particular sorting requirements, but it's a step in the right direction. Another step is to use the `I18N::Collate` module mentioned on page 242.

Modification with `\Q` and Friends: True Lies

Look in any documentation on Perl regular expressions, including the bottom of Table 7-1, and you are likely to find `\L`, `\E`, `\u`, and the other items listed here in Table 7-8. Yet, it might surprise you that they are not *really* regular-expression metacharacters. The regex engine understands that '`*`' means "any number" and that '`[`' begins a character class, but it knows nothing about '`\E`'. So why have I included them here?

Table 7-8: String and Regex-Operand Case-Modification Constructs

In-string Construct	Meaning	Built-in Function
`\L`, `\U`	lower, raise case of text until `\E` ✩	`lc (...)`, `uc(...)`
`\l`, `\u`	lower, raise case of next character †	`lcfirst (...)`, `ucfirst (...)`
`\Q`	add an escape before all non-alphabetic until `\E`	`quotemeta(...)`
	Special Combinations	
`\u \L`	Raise case of first character; lower rest until `\E` or end of text	
`\l \U`	Lower case of first character; raise rest until `\E` or end of text	

✩ In all these `\E` cases, "until the end of the string or regex" applies if no `\E` is given.

† In Perl5, ignored within `\L...\E` and `\U...\E` unless right after the `\L` or `\U`.

For most practical purposes, they appear to be normal regex metacharacters. When used in a regex *operand*, the doublequotish processing of Figure 7-1's Phase B handles them, so they normally never reach the regex engine (☞ 222). But because of cases where it does matter, I call these Table 7-8 items *second-class metacharacters.*

Second-class metacharacters

As far as regex-operands are concerned, variable interpolation and the metacharacters in Table 7-8 are special because they:

- are effective only during the "top-level" scan of the operand
- affect only characters in the compiled regular expression

This is because they are recognized only during Phase B of Figure 7-1, not later after the interpolation has taken place. If you don't understand this, you would be confused when the following didn't work:

```
$ForceCase = $WantUpper ? '\U' : '\L';
if (m/$ForceCase$RestOfRegex/) {
          ⋮
```

Because the `\U` or `\L` of `$ForceCase` are *in* the interpolated text (which is not processed further until Phase C), nothing recognizes them as special. Well, the regex engine recognizes the backslash, as always, so ⌜`\U`⌟ is treated as the general case of an unknown escape: the escape is simply ignored. If `$RestOfRegex` contains `Path` and `$WantUpper` is true, the search would be for the literal text `UPath`, not `PATH` as had been desired.

Another effect of the "one-level" rule is that something like ⌜`m/([a-z])…\U\1`⌟ doesn't work. Ostensibly, the goal is to match a lowercase letter, eventually followed by an uppercase version of the same letter. But `\U` works only on the text in the regular expression itself, and ⌜`\1`⌟ represents text *matched* by (some other part of the) expression, and that's not known until the match attempt is carried out. (I suppose a match attempt can be considered a "Phase E" of Figure 7-1.)

The Match Operator

The basic match is the core of Perl regular-expression use. Fortunately, the match operator is quite flexible; unfortunately, that makes mastering it more complex. In looking at what `m/…/` offers, I take a divide-and-conquer approach, looking at:

- Specifying the match operator itself (the regex operand)
- Specifying the target string to match against (the target operand)
- The match's side effects
- The value returned by a match
- Outside influences that affect the match

The Perl regular-expression match is an *operator* that takes two *operands* (a target string operand and a regex operand) and returns a value, although exactly what kind of value depends on context. Also, there are optional *modifiers* which change how the match is done. (Actually, I suppose these can be considered operands as well.)

Match-Operand Delimiters

It's common to use a slash as the match-operand delimiter, but you can choose any symbol you like. (Specifically, any non-alphanumeric, non-whitespace character may be used.*[6☞306]*) This is perhaps one of the most bizarre aspects of Perl syntax, although arguably one of the most useful for creating readable programs.

For example, to apply the expression ⌈`^/(?:[^/]+/)+Perl$`⌋ using the match operator with standard delimiters, `m/^\/(?:[^\/+\/)+Perl$/` is required. As presented in Phase A of Figure 7-1, a closing delimiter that appears within the regular expression must be escaped to hide the character's delimiter status. (These escapes are bold in the example.) Characters escaped for this reason are passed through to the regex as if no escape were present.*[7☞306]* Rather than suffer from backslashitis, it's more readable to use a different delimiter—two examples of the same regex are **`m!`**`^/(?:[^/]+/)+Perl$`**`!`** and **`m,`**`^/(?:[^/]+/)+Perl$`**`,`**. Other common delimiters are `m|…|`, `m#…#`, and `m%…%`.

There are several special-case delimiters:

- The four special cases `m(…)`, `m{…}`, `m[…]`, and `m<…>` have different opening and closing delimiters, and may be nested.*[8☞306]* Because parentheses and square brackets are so prevalent in regular expressions, `m(…)` and `m[…]` are probably not so appealing, but the other two can be. In particular, with the `/x` modifier, something such as the following becomes possible:

  ```
  m{
      regex   # comments
      here    # here
  }x;
  ```

  When these special-case delimiters are nested, matching pairs can appear unescaped. For example, **`m(^/((?:[^/]+/)+)Perl$)`** is valid, although visually confusing.

- If a singlequote is used as the delimiter, the regex operand undergoes *singlequotish interpolation,* as opposed to the normal doublequotish interpolation discussed in detail throughout this chapter. This means that Figure 7-1's Phase B is skipped, turning off variable interpolation and rendering inactive the items listed the Table 7-8. This could be useful if an expression has many `$` and `@` whose escaping becomes too untidy.

- If question marks are used as the delimiter,*[9☞306]* a very special kind of match operation is done. Normally, the match operator returns success whenever its regex operand can match its target string operand, but with this special `?`-delimited match, it returns success only with the first successful match. Subsequent uses fail, even if they could otherwise match, until `reset` in the same package is called.*[10☞306]* With a list-context `/g` match, "returning only once" still means "all the matches" as normal.

This can be useful if you want a particular match to be successful only once during a loop. For example, when processing an email header, you could use:

```
$subject = $1 if m?^Subject: (.*)?;
```

Once you've matched the subject once, there's no reason to bother checking for it on every line that follows. Still, realize that if, for whatever reason, the header has more than one subject line, `m?…?` matches only the first, while `m/…/` matches them all once the header has been processed, `m?...?` will have left `$subject` with the data from the first subject line, while `m/…/` will have left it with data from the last.

The substitution operator `s/…/…/` has other special delimiters that we'll talk about later ☞ 255; the above are the special cases for the match operator.

As yet another special case, if the delimiter is either the commonly used slash, or the special match-only-once question mark, the `m` itself becomes optional. It's common to use `/…/` for matches.

Finally, the generic pattern *target* `=~` *expression* (with no delimiters and no `m`) is supported. The expression is evaluated as a generic Perl expression, taken as a string, and finally fed to the regex engine. This allows you to use something like **`$text =~ &GetRegex()`** instead of the longer:

```
my $temp_regex = &GetRegex();
…$text =~ m/$temp_regex/…
```

Similarly, using **`$text =~ "`**…***string***…**`"`** could be useful if you wanted *real* doublequote processing, rather than the doublequote*ish* processing discussed earlier. But frankly, I would leave this to an Obfuscated Perl Contest.

The default regex

If no regex is given, such as with `m//` (or with `m/$regex/` where the variable `$regex` is empty or undefined), Perl reuses the regular expression *most recently used successfully within the enclosing dynamic scope.[11☞ 306]* In this case, any match modifiers (discussed in the next section) are completely ignored. This includes even the `/g` and `/i` modifiers. The modifiers used with the default expression remain in effect.

The default regex is never recompiled (even if the original had been built via variable interpolation without the `/o` modifier). This can be used to your advantage for creating efficient tests. There's an example in "The `/o` Modifier" (☞ 270).

Match Modifiers

The match operator supports a number of *modifiers* (options), which influence:

- how the regex operand is to be interpreted (`/o` ☞ 268; `/x` ☞ 223)
- how the regex engine considers the target text (`/i` ☞ 42; `/m`, `/s` ☞ 233)
- how the engine applies the regex (`/g` ☞ 253)

You can group several modifier letters together and place them in any order after the closing delimiter,* whatever it might be. For example, **`m/<code>/i`** applies the regular expression ⌈`<code>`⌋ with the `/i` modifier, resulting in a case-insensitive match. Do keep in mind that the slash is not part of the modifier—you could write this example as **`m|<code>|i`** or perhaps **`m{<code>}i`** or even **`m<<code>>i`**. As discussed earlier (☞ 231), the modifiers `/x`, `/i`, `/m`, and `/s` can also appear within a regular expression itself using the ⌈`(?…)`⌋ construct. Allowing these options to be indicated in the regular expression directly is extremely convenient when one operator is applying different expressions at different times (usually due to variable interpolation). When you use `/i`, for example, *every* application of an expression via the match operator in question is case-insensitive. By allowing each regular expression to choose its own options, you get more general-purpose code.

A great example is a search engine on a Web page that offers a full Perl regular expression lookup. Most search engines offer very simple search specifications that leaves power users frustrated, so offering a full regex option is appealing. In such a case, the user could use ⌈`(?i)`⌋ and the like, as needed, without the CGI having to provide special options to activate these modifiers.

`m/…/g` with a regex that can match nothingness

Normally, subsequent match attempts via the `/g` modifier start where the previous match ended, but what if there is a way for the regex to match the null string? As a simple example, consider the admittedly silly `m/^/g`. It matches at the start of the string, but doesn't actually consume any characters, so the first match ends at the beginning of the string. If the next attempt starts there as well, it will match there as well. Repeat forever and you start to see a problem.

Perl version 5.000 is broken in this respect, and indeed repeats until you run out of electrons. Perl4 and later versions of Perl5 work, although differently. They both begin the match where the previous one left off *unless* the previous match was of no text, in which case a special bump-along happens and the match is re-applied one character further down. Thus, each match after the first is guaranteed to progress down the string at least one character, and the infinite loop is avoided.

---

* Because match-operator modifiers can appear in any order, a large portion of a programmers time is spent adjusting the order to achieve maximal cuteness. For example, `learn/by/osmosis` is valid code (assuming you have a function called `learn`). The `o s m o s i s` are the modifiers—repetition of match-operator modifiers (but not the substitution-operator's `/e`) is allowed, but meaningless.

Except when used with the substitution operator, Perl5 takes the additional step of disallowing any match that ends at the same position as the previous match—in such a case the automatic one-character heave-ho is done (if not already at the end of the string). This difference from Perl4 can be important—Table 7-9 shows a few simple examples. (This table is not light reading by any means—it might take a while to absorb.) Things are quite different with the substitution operator, but that's saved for "The Substitution Operator" (☛ 255).

Table 7-9: Examples of `m.../g` with a Can-Match-Nothing Regex

regex:	⌈`\d*`⌋	match count	⌈`\d*`⌋	match count	⌈`x|\d*`⌋	match count	⌈`\d*|x`⌋*	match count
`Perl4`	123‸	2	123‸x‸	3	‸a123‸wx‸y‸z456‸	8	‸a123‸w‸x‸y‸z456‸	8
`Perl5`†	123	1	123x‸	2	‸a123wxy‸z456	5	‸a123w‸x‸y‸z456	6

(*Each match shown via either an underline, or as for a zero-width match*)

* As an aside, note that since ⌈`\d*`⌋ can never fail, the ⌈`|x`⌋ is never used, and so is meaningless.

† From version 5.001 on.

Specifying the Match Target Operand

Fortunately, the target string operand is simpler to describe than the regex operand. The normal way to indicate "This is the string to search" is using =~, as with **`$line =~ m/.../`**" Remember that =~ is *not* an assignment operator, nor is it a comparison operator. It is merely an odd way of providing the match operator with one of its operands. (The notation was adapted from *awk*)

Since the whole "***expr*** **`=~ m/.../`**" is an expression itself, you can use it wherever an expression is allowed. Some examples (each separated by a wavy line):

```
$text =~  m/.../;     #  just do it, presumably, for the side effects.
```

* * *

```
if ($text =~ m/.../) {
   ##  do code if match successful

    ⋮
```

* * *

```
$result = ( $text   =~ m/.../ );  #  set $result to result of match against $text
$result =   $text   =~ m/.../  ;  #  same thing =~ has higher precedence than =
```

* * *

```
$result =   $text;                  #  copy $text to $result...
$result            =~ m/.../   ;    #  . . . and perform match on result
( $result =  $text ) =~ m/.../   ;  #  Same thing in one expression
```

If the target operand is the variable $_, you can omit the "**$_=~**" altogether. In other words, the default target operand is $_.

Something like **`$line =`**~ **`m/`*****regex*****`/`** means "Apply *regex* to the text in `$line`, ignoring the return value but doing the side effects." If you forget the '~', the resulting **`$line = m/`*****regex*****`/`** becomes "Apply *regex* to the text in `$_`, returning a true or false value that is then assigned to `$line`." In other words, the following are the same:

```
$line =           m/regex/
$line = ($_ =~   m/regex/)
```

You can also use `!~` instead of `=~` to logically negate the return value. (Return values and side effects are discussed soon.) **`$var !~ m/…/`** is effectively the same as **`not ($var =~ m/…/)`**. All the normal side effects, such as the setting of `$1` and the like, still happen. It is merely a convenience in an "If this doesn't match" situation. Although you can use `!~` in a list context, it doesn't make much sense.

Other Side Effects of the Match Operator

Often, more important than the actual return value of the match are the resulting side effects. In fact, it is quite common to use the match operator without using its return value just to obtain the side effects. (The default context is scalar.) I've already discussed most of the side effects (`$&`, `$1`, `$+`, and so on; ☞ 217), so here I'll look at the remaining side effects a match attempt can have.

Two involve "invisible status." First, if the match is specified using `m?…?`, a successful match dooms future matches to fail, at least until the next reset (☞ 247). Of course, if you use `m?…?` explicitly, this particular side effect is probably the main effect desired. Second, the regex in question becomes the default regex until the dynamic scope ends, or another regex matches (☞ 248).

Finally, for matches with the `/g` modifier, the `pos` of the target string is updated to reflect the index into the string of the match's end. (A failed attempt always resets pos.) The next `/g`-governed match attempt of the same string starts at that position, unless:

- the string is modified. (Modifying a string resets its pos.)
- `pos` is assigned to. (The match will start at that position.)
- the previous match did not match at least one character.

As discussed earlier (☞ 249), in order to avoid an infinite loop, a successful match that doesn't actually match characters causes the next match to begin one character further into the string. During the begun-further attempt, `pos` properly reflects the end of the match because it's at the start of the subsequent attempt that the anti-loop movement is done. During such a next match, ⌜`\G`⌟ still refers to the true end of the previous match, so cannot be successful. (This is the only situation where `\G` doesn't mean "the start of the attempt.")

Match Operator Return Value

Far more than a simple true/false, the match operator can return a wide variety of information. (It can also return a simple true/false value if you like.) The exact information and the way it's returned depends on two main factors: *context* and the `/g` modifier.

Scalar context, without the `/g` modifier

Scalar context without the `/g` modifier is the "normal situation." If a match is found, a Boolean true is returned:

```
    if ($target =~ m/.../) {
        #   processing if match found
              ⋮
    } else {
        #   processing if no match found
              ⋮
    }
```

On failure, it returns an empty string (which is considered a Boolean false).

List context, without the `/g` modifier

A list context without `/g` is the normal way to pluck information from a string. The return value is a list with an element for each set of capturing parentheses in the regex. A simple example is processing a date of the form `69/8/31`, using:

```
     ($year, $month, $day) = $date =~ m{^ (\d+) / (\d+) / (\d+)
$}x;
```

The three matched numbers are then available in the three variables (and `$1` and such as well).*[12☞ 307]* There is one element in the return-value list for each set of capturing parentheses, or an empty list upon failure. Of course, it is possible for a set or sets to have not been part of a match, as is certainly guaranteed with one in **`m/(this)|(that)/`**. List elements for such sets exist, but are undefined.*[13☞ 307]* If there are no sets of capturing parentheses to begin with, a list-context nor-`/g` match returns the list (1).

Expanding a bit on the date example, using a match expression as the conditional of an `if (…)` can be useful. Because of the assignment to (`$year, …`), the match operator finds itself in a list context and returns the values for the variables. But since that whole assignment expression is used in the scalar context of the if's conditional, it is then contorted into the count of items in the list. Conveniently, this is interpreted as a Boolean false if there were no matches, true if there were.

```
    if ( ($year, $month, $day) = $date =~ m{^ (\d+) / (\d+) /
(\d+) $}x ) {
        # Process for when we have a match: $year and such have new values
    } else {
        # Process for when no match: $year and such have been newly cleared to
undefined
    }
```

List context, with the `/g` modifier

Available since version 4.036, this useful construct returns a list of all text matched within capturing parentheses (or if there are no capturing parentheses, the text matched by the whole expression), not only for one match as with the non-`/g` list-context, but for all matches in the string. For example, consider having the entire text of a Unix mailbox alias file in a single string, where logical lines look like:

```
alias      jeff      jfriedl@ora.com
alias      perlbug   perl5-porters@perl.org
alias      prez      president@whitehouse
```

You can use something like `m/^alias\s+(\S+)\s+(.+)/` to pluck the alias and full address from a single logical line. It returns a list of two elements, such as `('jeff', 'jfriedl@ora.com')` for the first line. Now consider working with all the lines in one string. You can do all the matches all at once by using `/g` (and `/m`, to allow caret to match at the beginning of each logical line), returning a list such as:

```
      ( 'jeff', 'jfriedl@ora.com', 'perlbug',
        'perl5-porters@perl.org', 'prez', 'president@whitehouse'
)
```

If it happens that the items returned fit the key/value pair pattern as in this example, you can actually assign it directly to an associative array. After running

```
    %alias = $text =~ m/^alias\s+(\S+)\s+(.+)/mg;
```

you could access the full address of `'jeff'` with `$alias{'jeff'}`.

Scalar context, with the `/g` modifier

A scalar-context `m/…/g` is a special construct quite different from the other three situations. Like a normal `m/…/`, it does only one match, but like a list-context `m/…/g`, it pays attention to where previous matches occurred. Each time a scalar-context `m/…/g` is reached, such as in a loop, it finds the "next" match. Once it fails, the next check starts again from the beginning of the string.

This is quite convenient as the conditional of a `while` loop. Consider:

```
while ($ConfigData =~ m/^(\w+)=(.*)/mg){
    my($key, $value) = ($1, $2);
      ⋮
}
```

All matches are eventually found, but the body of the `while` loop is executed between the matches (well, *after* each match). Once an attempt fails, the result is false and the `while` loop finishes. Also, upon failure, the `/g` state (given by `pos`) is reset. Finally, be careful not to modify the target data within the loop unless you really know what you're doing: it resets `pos`.*[14¬307]*

Outside Influences on the Match Operator

We've just spent a fair amount of time covering the various options, special cases, and side effects of the match operator. Since many of the influences are not visually connected with the application of a match operator (that is, they are used or noticed elsewhere in the program), I'd like to summarize the hidden effects:

- *context* Has a large influence on how the match is performed, as well as on its return value and side effects.

- `pos (…)` Set explicitly, or implicitly by the `/g` modifier, it indicates where in the string the next `/g`-governed match should begin. Also, see 「`\G`」 in "MultiMatch Anchor" (☞ 236).

- `$*` A holdover from Perl4, can influence the caret and dollar anchors. See "String Anchors" (☞ 232).

- *default regex* Is used if the provided regex is empty (☞ 248).

- `study` Has no effect on what is matched or returned, but if the target string has been `study'd`, the match might be faster (or slower). See "The Study Function" (☞ 287).

- `m?…?/reset` affects the invisible "has/hasn't matched" status of `m?…?` operators (☞ 247).

Keeping your mind in context (and context in mind)

Before leaving the match operator, I'll put a question to you. Particularly when changing among the `while`, `if`, and `foreach` control constructs, you really need to keep your wits about you. What do you expect the following to print?

```
while ("Larry Curly Moe" =~ m/\w+/g) {
   print "WHILE stooge is $&.\n";
}
print "\n";

if ("Larry Curly Moe" =~ m/\w+/g)
   print "IF stooge is $&.\n";
}
print "\n";

foreach ("Larry Curly Moe" =~ m/\w+/g) {
   print "FOREACH stooge is $&.\n";
}
```

It's a bit tricky. ❖ Turn the page to check your answer.

The Substitution Operator

Perl's substitution operator `s`*/regex/replacement/* extends the idea of matching text to match-and-replace. The regex operand is the same as with the match operator, but the *replacement operand* used to replace matched text adds a new, useful twist. Most concerns of the substitution operator are shared with the match operator and are covered in that section (starting on page 246). New concerns include:

- The replacement operand and additional operand delimiters
- The `/e` modifier
- Context and return value
- Using `/g` with a regex that can match nothingness

The Replacement Operand

With the normal `s/…/…/`, the replacement operand immediately follows the regex operand, using a total of three instances of the delimiter rather than the two of `m/…/`. If the regex uses balanced delimiters (such as `<…>`), the replacement operand then has its own independent pair of delimiters (yielding four delimiters). In such cases, the two sets may be separated by whitespace, and if so, by comments as well.*[15☞307]* Balanced delimiters are commonly used with `/x` or `/e`:

```
$test =~ s{
…some big regex here, with lots of comments and such…
}   {
…a perl code snippet to be evaluated to produce the replacement text…
}ex
```

Perl normally provides true doublequoted processing of the replacement operand, although there are a few special-case delimiters. The processing happens after the match (with `/g`, after each match), so `$1` and the like are available to refer to the proper match slice.

Special delimiters of the replacement operand are:

- As with the match operator, you can use singlequotes for the regex operand as well,*[16☞307]* but that's almost a separate issue. (If you use them for the regex operand, you must use them for the replacement operand, but not vice-versa.)

- The special match-operator once-only delimiter `?…?` is not special with the substitution operator.

- Despite previous Perl5 documentation to the contrary, the use of backquotes as the delimiter is *not* special as it was in Perl4. In Perl4, the replacement operand first underwent doublequote interpolation, the results of which were executed as a shell command. The resulting output was then used as the

`while` *vs.* `foreach` *vs.* `if`

❖ Answer to the question on page 254.

The results differ depending on your version of Perl:

	Perl4	**Perl5**	
	`WHILE stooge is Larry.`	`WHILE stooge is Larry.`	
	`WHILE stooge is Curly.`	`WHILE stooge is Curly.`	
	`WHILE stooge is Moe.`	`WHILE stooge is Moe.`	
	`IF stooge is Larry.`	`IF stooge is Larry.`	
	`FOREACH stooge is`	`FOREACH stooge is Moe.`	
	`FOREACH stooge is`	`FOREACH stooge is Moe.`	
	`FOREACH stooge is .`	`FOREACH stooge is Moe.`	

Note that if the `print` within the `foreach` loop had referred to `$_` rather than `$&`, its results would have been the same as the `while`'s. In this `foreach` case, however, the result returned by the `m/…/g`, **(**`'Larry', 'Curly', 'Moe'`**)**, go unused. Rather, the side effect `$&` is used, which almost certainly indicates a programming mistake, as the side effects of a list-context `m/…/g` are not often useful.

replacement text. When needed (rarely), this behavior can be closely mimicked in Perl5. Compare the following for matching command names and fetching their **-**version strings:

Perl4 : `s' version of (\w+) '$1 --version 2>&l 'g;`
Both Perl4 and Perl5: `s/ version of (\w+) /'$1 --version 2>&1'/e;`

The marked portion is the replacement operand. In the first version, it is executed as a system command due to the special delimiter. In the second version, the backquotes are not special until the whole replacement is `eval`uated due to the `/e` modifier. The `/e` modifier, discussed in detail in just a moment, indicates that the replacement operand is a mini Perl snippet that should be executed, and whose final value should be used as the replacement text.

Remember that this replacement-operand processing is all quite distinct from regex-operand processing, which usually gets doublequotish processing and has its own set of special-case delimiters.

The /e Modifier

Only the substitution operator allows the use of `/e` modifier. When used, the replacement operand is evaluated as if with **`eval {…}`** (including the load-time syntax check), the result of which is substituted for the matched text. The replacement operand does not undergo any processing before the `eval` (except to determine its lexical extent, as outlined in Figure 7-1's Phase A), not even singlequotish processing.*[17☞308]* The actual evaluation, however, is redone upon each match.

As an example, you can encode special characters of a World Wide Web `URL` using `%` followed by their two-digit hexadecimal representation. To encode all non-alphanumerics this way, you can use

```
    $url =~ s/([^a-zA-Z0-9])/sprintf('%%%02x', ord($1))/ge;
```

and to decode it back, you can use:

```
    $url =~ s/%([0-9a-f][0-9a-f])/pack("C",hex($1))/ige;
```

In short, **`pack("C",`** ***value*****`)`** converts from a numeric value to the character with that value, while **`sprintf('%%%02x', ord(`*****character*****`))`** does the opposite; see your favorite Perl documentation for more information. (Also, see the footnote on page 66 for more on this example.)

The moving target of interpretation

Particularly with the `/e` modifier, you should understand exactly *who* interprets *what* and *when.* It's not too confusing, but it does take some effort to keep things straight. For example, even with something as simple as **`s/…/'echo $$'/e`**, the question arises whether it's Perl or the shell that interprets the `$$`. To Perl and many shells, `$$` is the process `ID` (of Perl or the shell, as the case may be). You must consider several levels of interpretation. First, the replacement-operand has no `pre-eval` processing in Perl5, but in Perl4 has singlequotish processing. When the result is `evaluated`, the backquotes provide doublequoted-string processing. (This is when Perl interpolates `$$` it may be escaped to prevent this interpolation.) Finally, the result is sent to the shell, which then runs the *echo* command. (If the `$$` had been escaped, it would be passed to the shell unescaped, resulting in the shell's interpolation of `$$`.)

To add to the fray, if using the `/g` modifier as well, should `'echo $$'` be evaluated just once (with the result being used for all replacements), or should it be done after each match? When `$1` and such appear in the replacement operand, it obviously must be evaluated on a per-match basis so that the `$1` properly reflects its after-match status. Other situations are less clear. With this `echo` example, Perl version 5.000 does only one evaluation, while other versions before and after evaluate on a per-match basis.

/eieio

Perhaps useful only in an Obfuscated Perl Contest, it is interesting to note that the replacement operand will be `eval`uated multiple times if `/e` is specified more than once. (It is the only modifier for which repetition matters.) This is what Larry 'Wall calls an *accidental feature,* and was "discovered" in early 1991. During the ensuing `comp.lang.perl` discussion, Randal Schwartz offered one of his patented JAPH signatures:[*]

```
   $Old_MacDonald = q#print #; $had_a_farm = (q-q:Just another
Perl hacker,:-);
   s/^/q[Sing it, boys and
girls…],$Old_MacDonald.$had_a_farm/eieio;
```

The `eval` due to the first `/e` sees

```
    q[Sing it, boys and girls…],$Old_MacDonald.$had_a_farm
```

whose execution results in `print q:Just another Perl hacker,:` which then prints Randal's "Just another Perl hacker" signature when evaluated due to the second `/e`.

Actually, this kind of construct is sometimes useful. Consider wanting to interpolate variables into a string manually (such as if the string is read from a configuration file). A simple approach uses **`$data =~ s/(\$[a-zA-Z_]\w*)/$1/eeg;`**. Applying this to '`option=$var`', the regex matches `option=`<u>`$var`</u>. The first `eval` simply sees the snippet `$1` as provided in the replacement-operand, which in this case expands to `$var`. Due to the second `/e`, this result is evaluated again, resulting in whatever value the variable `$var` has at the time. This then replaces the matched '`$var`', in effect interpolating the variable.

I actually use something like this with my personal Web pages most of them are written in a pseudo Perl/HTML code that gets run through a CGI when pulled by a remote client. It allows me to calculate things on the fly, such as to remind readers how few shopping days are left until my birthday.[**]

Context and Return Value

Recall that the match operator returns different values based upon the particular combination of context and `/g`. The substitution operator, however, has none of these complexities—it returns the same type of information regardless of either concern.

The return value is either the number of substitutions performed or, if none were done, an empty string.*[18☞308]* When interpreted as a Boolean (such as for the conditional of an `if`), the return value conveniently interprets as true if any substitutions were done, false if not.

---

* My thanks to Hans Mulder for providing the historical background for this section, and for Randal for being Just Another Perl Hacker with a sense of humor.

** If you, too, would like to see how many days are left until my birthday, just load
`http://omrongw.wg.omron.co.jp/cgi-bin/j-e/jfriedl.html`
or perhaps one of its mirrors (see Appendix A).

Using /g with a Regex That Can Match Nothingness

The earlier section on the match operator presented a detailed look at the special concerns when using a regex that can match nothingness. Different versions of Perl act differently, which muddies the water considerably. Fortunately, all versions' substitution works the same in this respect. Everything presented earlier in Table 7-9 (☞ 250) applies to how `s/…/…/g` matches as well, but the entries marked "Perl5" are for the match operator only. The entries marked "Perl4" apply to Perl4's match operator, and all versions' substitution operator.

The Split Operator

The multifaceted `split` operator (often called a *function* in casual conversation) is commonly used as the converse of a list-context `m/…/g` (☞ 253). The latter returns text matched by the regex, while a `split` with the same regex returns text *separated* by matches. The normal match **`$text =~ m/:/g`** applied against a `$text` of '`IO.SYS:225558:95-10-03:-a-sh:optional`', returns the four-element list

```
(':', ':', ':', ':')
```

which doesn't seem useful. On the other hand, **`split(/:/, $text)`** returns the five-element list:

```
('IO.SYS',  '225558',  '95-10-03',  '-a-sh',  'optional')
```

Both examples reflect that ⌈`:`⌋ matches four times. With `split`, those four matches partition a copy of the target into five chunks which are returned as a list of five strings.

In its most simple form with simple data like this, `split` is as easy to understand as it is useful. However, when the use of `split` or the data are complicated, understanding is less clear-cut. First, I'll quickly cover some of the basics.

Basic Split

split is an operator that looks like a function, and takes up to three operands:

```
    split(match operand, target string, chunk-limit operand)
```

(The parentheses are optional with Perl5.) Default values, discussed below, are provided for operands left off the end.

Basic match operand

The match operand has several special-case situations, but normally it's is a simple regular expression match such as `/:/` or `m/\s*<P>\s*/i`. Conventionally, `/…/`

rather than `m/…/` is used, although it doesn't really matter. The `/g` modifier is not needed (and is ignored) because `split` itself provides the iteration for matching in multiple places.

There is a default match operand if one is not provided, but it is one of the complex special cases discussed later.

Target string operand

The target string is inspected, and is never modified by `split`. The content of `$_` is the default if no target string is provided.

Basic chunk-limit operand

In its primary role, the chunk-limit operand specifies a limit to the number of chunks that split partitions the string into. For example, with our sample data, `split(/:/, $text, 3)` returns:

```
( 'IO.SYS',  '225558',  '95-10-03:-a-sh:optional'
```

This shows that `split` stopped after `/:/` matched twice, resulting in the requested three-chunk partition. It could have matched additional times, but that's irrelevant here because of the chunk-limit. The limit is an upper bound, so no more than that many elements will ever be returned, but note that it doesn't guarantee that many elements—no extra are produced to "fill the count" if the data can't be partitioned enough to begin with. **`split(/:/, $text, 1234)`** still returns only a five-element list. Still, there is an important difference between **`split(/:/, $text)`** and **`split(/:/, $text, 1234)`** which does not manifest itself with this example   keep this in mind for when the details are discussed later.

Remember that the *chunk*-limit operand is not a *match*-limit operand. Had it been for the example above, the three *matches* would have partitioned to

```
('IO.SYS',  '225558',  '95-10-03',  '-a-sh:optional')
```

which is not what actually happens.

One comment on efficiency: Let's say you intended to fetch only the first few fields, such as with

```
($filename, $size, $date) = split(/:/, $text)
```

you need *four* chunks—filename, size, date, and "everything else." You don't even want the "everything else" except that if it weren't a separate chunk, the `$date` chunk would contain it. So, you'd want to use a limit of `4` so that Perl doesn't waste time finding further partitions. Indeed, you can use a chunk-limit of `4`, but if you don't provide one, Perl provides an appropriate default so that you get the performance enhancement without changing the results you actually see.*[19☞ 308]*

Advanced Split

Because `split` is an operator and not a function, it can interpret its operands in magical ways not restricted by normal function-calling conventions. That's why, for example, `split` can recognize whether the first operand is a match operator, as opposed to some general expression that gets evaluated independently before the "function" is called.

Although useful, `split` is not straightforward to master. Some important points to consider are:

- split's match operand differs from the normal `m/…/` match operator. In addition, there are several special-case match operands.
- When the match operand matches at the beginning or end of the target string, or twice in a row, it usually returns empty strings on one side or the other. Usually, but not always.
- What happens when the regex can match nothingness?
- `split` behaves differently when its regex has capturing parentheses.
- A scalar-context `split`, now deprecated, stuffs the list into `@_` instead of returning it.

Split can return empty elements

The basic premise of `split` is that it returns the text between matches. If the match operator matches twice in a row, the nothingness between the matches is returned. Applying `m/:/` to the sample string

```
:IO.SYS:225558:::95-10-03:-a-sh:
```

finds seven matches (each marked). With `split`, a match always[*] results in a split between *two* items, including even that first match at the start of the string separating `''` (an empty string, i.e., nothingness) from '`IO.SY`…'. Similarly, the fourth match separates two empty strings. All in all, the seven matches partition the target into the eight strings:

```
     ( '', 'IO.SYS', '225558', '', '', '95-10-03', '-a-sh', ' ' )
```

However, this is *not* what `split(/:/, $text)` returns. Surprised?

Trailing empty elements are not (normally) returned

When the chunk-limit operand is not specified, as it often isn't, Perl strips trailing empty items from the list before it is returned. (Why? Your guess is as good as mine, but the feature is indeed documented.) Only empty items at the end of the list are stripped; others remain. You can, however, stop Perl from removing the trailing empty items, and it involves a special use of the chunk-limit operand.

---

[*] There is one special case where this is not true. It is detailed a bit later during the discussion about advanced `split`'s match operand.

The chunk-limit operand's second job

In addition to possibly limiting the number of chunks, any non-zero chunk-limit operand also eliminates the stripping of trailing empty items. (A chunk-limit given as zero is exactly the same as if no chunk limit is given at all.)

If you don't want to limit the number of chunks returned, but instead only want to leave trailing empty items intact, simply choose a very large limit. Also, a negative chunk-limit is taken as an arbitrarily large limit: **`split(/:/, $text, -1)`** returns all elements, including any trailing empty ones.

At the other extreme, if you want to remove *all* empty items, you could put **`grep {length}`** before the `split`. The `grep` lets pass only list elements with nonzero lengths (in other words, elements that aren't empty).

Advanced Split's Match Operand

Most of the complexity of the `split` operator lies in the many personalities of the match operand. There are four distinct styles of the match operand:

- a match(-like) operator
- the special scalar `' '` (a single space)
- any general scalar expression
- the default for when no operand is provided

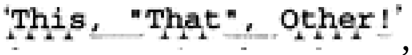
Split's match-operator match operand

Like all the `split` examples we've seen so far, the most common use of `split` uses a match operator for the match operand. However, there are a number of important differences between a match operand and a real match operator:

- You should not (and in some versions of Perl, cannot) specify the match operator's target string operand. Never use `=~` with `split`, as strange things can happen. `split` provides the match operand with the target string operand behind the scenes.

- We've discussed the complexity of a regex that can match nothingness, such as with `m/x*/` (☞ 249) and `s/x*/…/` (☞ 259). Perhaps because `split` provides the repetition instead of the match operator, things are much simpler here.

  The one exception is that a match of nothingness at the start of the string does *not* produce a leading empty element. Contrast this to a match of *something* at the start of the string: it does produce a leading empty element. (A match of nothingness at the end of the string does leave a trailing one, though, but such trailing empty items are removed unless a large enough, or negative, chunk-limit is used.)

For example, `m/\W*/` matches  'This, "That", Other!', Other!' at the places marked (either underlined, or withfor a match of nothingness). Because of this exception, the string-leading match is ignored, leaving 13 matches, resulting in the 14 element list:

```
      ( 'T', 'h', 'i', 's', 'T', 'h', 'a', 't', 'o', 't', 'h',
'e', 'r', '')
```

Of course, if no chunk-limit is specified, the final empty element is removed.

- An empty regex for `split` does not mean "Use the current default regex," but to split the target string at each character. For example, you could use

```
$text = join "\b_", split(//, $text, -1);
```

to put a backspace-underline sequence after each character in `$text`. (However, **`$text =~ s/(.)/$1\b_/g`** might be better for a variety of reasons, one of which is that it's probably easier to understand at first glance).

- The use of a regex with `split` doesn't affect the default regex for later match and substitution operators. Also, the variables `$&`, `$'`, `$1`, and so on are not available from a `split`. A `split` is completely isolated from the rest of the program with respect to side-effects.

- The `/g` modifier is meaningless (but harmless) when used with `split`.

- The special regex delimiter `?…?` is not special with `split`. *[20☞ 308]*

Special match operand: a string with a single space

A match operand that is a *string* (not a regex) consisting of exactly one space is a special case. It's almost the same as `/\s+/` except that leading whitespace is skipped. (This is meant to simulate the default input-record-separator splitting that *awk* does with its input, although it can certainly be quite useful for general use.)

For instance, the call **`split('•', "•••this•••is•a•••••test")`** returns the four-element list **`('this', 'is', 'a', 'test')`**. As a contrast to `'•'`, consider using `m/\s+/` directly. This bypasses the leading-whitespace removal and returns **`('', 'this', 'is', 'a', 'test')`**

Finally, both of these are quite different from using `m/•/`, which matches each individual space and returns:

```
        ('','','','this','','','is','a','','','','test')
```

Any general scalar expression as the match operand

Any other general Perl expression as the match operand is evaluated independently, taken as a string, and interpreted as a regular expression. For example, `split(/\s+/, …)` is the same as `split('\s+', …)` except the former's regex is compiled only once, the latter's each time the `split` is executed. *[21☞308]*

The default match operand

With Perl5, the default match operand when none is specified (which is different from `//` or `''`), is identical to using `'•'`. Thus, a raw `split` without any operands is the same as `split('•', $_, 0)`. *[22☞308]*

Scalar-Context Split

Perl4 supported a scalar-context `split`, which returned the number of chunks instead of the list itself, causing the variable `@_` to receive the list of chunks as a side effect. Although Perl5 currently supports it, this feature has been deprecated and will likely disappear in the future. Its use generates a warning when warnings are enabled, as they generally should be.

Split's Match Operand with Capturing Parentheses

Using capturing parentheses in the match-operand regex changes the whole face of `split`. In such a case, the returned array has additional, independent elements interjected for the item(s) captured by the parentheses. This means that text normally elided entirely by `split` is now included in the returned list.

In Perl4, this was more of a pain than a useful feature because it meant you could never (easily) use a regex that happened to require parentheses merely for grouping. If grouping is the only intent, the littering of extra elements in the return list is definitely not a feature. Now that you can select the style of parentheses capturing or not it's a great feature. For example, as part of HTML processing, `split(/(<[^>]*>)/)` turns

```
    ...•and•<B>very•<FONT•color=red>very</FONT>•much</B>
•effort...
```

into

```
     ( '...•and ', '<B>', 'very•', '<FONT•color=red>',
       'very', '</FONT>', '•much', '</B>', '•effort...' )
```

which might be easier to deal with. This example so far works with Perl4 as well, but if you want to extend the regex to be cognizant of, say, simple ⌈`"[^"]*`⌋ doublequoted strings (probably necessary for real HTML work), you run into problems as the full regex becomes something along the lines of:*

⌈`(<[^>"]*("[^"]*"[^>"]*)*>)`⌋

The added set of capturing parentheses means an added element returned for each match during the split, yielding two items per match plus the normal items due to the split. Applying this to

```
Please <A HREF="test">press me</A> today
```

returns (with descriptive comments added):

```
( 'Please·',                                    before first match
  '<A·HREF="test">', '"test"',                  from first match
  'press me',                                   between matches
  '</A>', '',                                   from second match
  '·today'                                      after last match
)
```

The extra elements clutter the list. Using ⌈`(?:…)`⌋ for the added parentheses, however, returns the regex to `split` usefulness, with the results being:

```
( 'Please·',                                    before first match
  '<A HREF="test">',                            from first match
  'press me',                                   between matches
  '</A>',                                       from second match
  '·today'                                      after last match
)
```

Perl Efficiency Issues

For the most part, efficiency with Perl regular expressions is achieved in the same way as with any tool that uses a Traditional NFA: use the techniques discussed in Chapter 5—the internal optimizations, the unrolling methods, the "Think" section—they all apply to Perl.

There are, of course, Perl-specific efficiency issues, such as the use of non-capturing parentheses unless you specifically need capturing ones. There are some much larger issues as well, and even the issue of capturing *vs.* non-capturing is larger than the micro-optimization explained in Chapter 5 (☞ 152). In this section, we'll look at this (☞ 276), as well as the following topics:

- **There's More Than One Way To Do It** Perl is a toolbox offering many approaches to a solution. Knowing which problems are nails comes with understanding *The Perl Way*, and knowing which hammer to use for any particular nail goes a long way toward making more efficient and more understandable programs. Sometimes efficiency and understandability seem to be mutually exclusive, but a better understanding allows you to make better choices.

---

* You might recognize this as being a partially unrolled version of ⌈**`<("[^"]*"|[^>"])*>`**⌋, with a *normal* of ⌈`[^>"]`⌋ and a *special* of ⌈`"[^"]*"`⌋. You could, of course, also unroll *special* independently.

- **Interpolation** The interpolation and compilation of regex operands is fertile ground for saving time. The `/o` modifier, which I haven't discussed much yet, gives you some control over when the costly re-compilation takes place.

- **$& Penalty** The three match side effect variables, `` $` ``, $&, and `$'`, can be convenient, but there's a hidden efficiency *gotcha* waiting in store for any script that uses them, even once, anywhere. Heck, you don't even have to *use* them the entire script is penalized if one of these variables even *appears* in the script.

- **`/i` Penalty** You pay another penalty when you use the `/i` modifier. Particularly with very long target strings, it can pay to rewrite the regex to avoid using `/i`.

- **Substitution** You can get the most out of the various ways that Perl optimizes substitutions by knowing when the optimizations kick in, what they buy you, and what defeats them.

- **Benchmarking** When it comes down to it, the fastest program is the one that finishes first. (You can quote me on that.) Whether a small routine, a major function, or a whole program working with live data, benchmarking is the final word on speed. Benchmarking is easy and painless with Perl, although there are various ways to go about it. I'll show you the way I do it, a simple method that has served me well for the hundreds of benchmarks I've done while preparing this chapter.

- **`Study`** Since ages past, Perl has provided the `study(…)` function. Using it supposedly makes regexes faster, but no one really understands it. We'll see whether we can figure it out.

- **`-Dr` Option** Perl's regex-debug flag can tell you about some of the optimizations the regex engine and transmission do, or don't do, with your regexes. We'll look at how to do this and see what secrets Perl gives up.

"There's More Than One Way to Do It"

There are often many ways to go about solving any particular problem, so there's no substitute for really knowing all that Perl has to offer when balancing efficiency and readability. Let's look at the simple problem of padding an IP address like `18.181.0.24` such that each of the four parts becomes exactly three digits: `018.181.000.024`. One simple and readable solution is:

```
    $ip = sprintf "%03d.%03d.%03d.%03d", split(/\./, $ip);
```

This is a fine solution, but there are certainly other ways to do the job. In the same style as the *The Perl Journal* article I mentioned in the footnote on page 229, let's examine various ways of achieving the same goal. This example's goal is simple and not very "interesting" in and of itself, yet it represents a common text-handling task. Its simplicity will let us concentrate on the differing approaches to using Perl. Here are a few other solutions:

```
1.      $ip =~ s/(\d+)/sprintf("%03d", $1)/eg;
2.      $ip =~ s/\b(\d{1,2}\b)/sprintf("%03d", $1)/eg;
3.      $ip =  sprintf("%03d.%03d.%03d.%03d", $ip =~ m/(\d+)/g);
4.      $ip =~ s/\b(\d\d?\b)/'0' x (3-length($1)) . $1/eg;
5.      $ip =  sprintf("%03d.%03d.%03d.%03d",
                       $ip =~ m/^(\d+)\.(\d+)\.(\d+)\.(\d+)$/);
6.      $ip =~ s/\b(\d(\d?)\b)/$2 eq '' ? "00$1"  : "0$1"/eg;
7.      $ip =~ s/\b(\d\b)/00$1/g;
        $ip =~ s/\b(\d\d\b)/0$1/g;
```

Like the original solution, each produces the same results when given a correct IP address, but fail in different ways if given something else. If there is any chance that the data will be malformed, more care than any of these solutions provide is needed. That aside, the practical differences lie in efficiency and readability. As for readability, about the only thing that's easy to see about most of these is that they are cryptic at best.

So, what about efficiency? I benchmarked these solutions on my system with Perl version 5.003, and have listed them in order from least to most efficient. The original solution belongs somewhere between positions four and five, the best taking only 80 percent of its time, the worst about 160 percent.* But if efficiency is *really* important, faster methods are still available:

```
    substr($ip,      0, 0) = '0' if substr($ip,      1, 1) eq
'.';
    substr($ip,      0, 0) = '0' if substr($ip,      2, 1) eq
'.';
    substr($ip,      4, 0) = '0' if substr($ip,      5, 1) eq
'.';
    substr($ip,      4, 0) = '0' if substr($ip,      6, 1) eq
'.';
    substr($ip,      8, 0) = '0' if substr($ip,      9, 1) eq
'.';
    substr($ip,      8, 0) = '0' if substr($ip,     10, 1) eq
'.';
    substr($ip,     12, 0) = '0' while length($ip) < 15;
```

This takes only half the time as the original, but at a fairly expensive toll in understandability. Which solution you choose, if any, is up to you. There are probably other ways still. Remember, "There's more than one way to do it."

---

* With Perl4, for reasons I don't exactly know, the original solution is actually the fastest of those listed. I wasn't able, however, to benchmark the solutions using `/e` due to a related Perl4 memory leak that rendered the results meaningless.

Regex Compilation, the `/o` *Modifier, and Efficiency*

In "Doublequotish Processing and Variable Interpolation," we saw how Perl parses a script in phases. All the processing represented by Figure 7-1 (☞ 223) can be rather substantial. As an optimization, Perl realizes that if there is no variable interpolation, it would be useless to bother processing all the phases each time the regex is used if there is no interpolation, the regex can't possibly change from use to use. In such cases, the internal form is saved the first time the regex is compiled, then used directly for all subsequent matches via the same operator. This saves a lot of reprocessing work.

On the other hand, if the regex changes each time, it certainly makes sense for Perl to reprocess it for us. Of course, it takes longer to redo the processing, but it's a very convenient feature that adds remarkable flexibility to the language, allowing a regex to vary with each use. Still, as useful as it may be, the extra work is sometimes needless. Consider a situation where a variable that doesn't change from use to use is used to provide a regex:

```
        $today = (qw<Sun Mon Tue Wed Thu Fri
Sat>)[(localtime)[6]];
        #  $today now holds the day ("Mon", "Tue", etc., as appropriate)
        $regex = "^$today:";
        while (<LOGFILE>) {
            if (m/$regex/)  {
               ⋮
```

The variable `$regex` is set just once, before the loop. The match operator that uses it, however, is inside a loop, so it is applied over and over again, once per line of `<LOGFILE>`. *We* can look at this script and know for sure that the regex doesn't change during the course of the loop, but *Perl* doesn't know that. It knows that the regex operand involves interpolation, so it must re-evaluate that operand each time it is encountered.

This doesn't mean the regex must be fully recompiled each time. As an intermediate optimization, Perl uses the compiled form still available from the previous use (of the same match operand) *if* the re-evaluation produces the same final regex. This saves a full recompilation, but in cases where the regex never does change, the processing of Figure 7-1's Phase B and C, and the check to see if the result is the same as before, are all wasted effort.

This is where the `/o` modifier comes in. It instructs Perl to process and compile a regex operand the first time it is used, as normal, but to then blindly use the same internal form for all subsequent tests by the same operator. The `/o` "locks in" a regex the first time a match operator is used. Subsequent uses apply the same regex *even if variables making up the operand were to change*. Perl won't even bother looking. Normally, you use `/o` as a measure of efficiency when you don't intend to change the regex, but you must realize that even if the variables *do* change, by design or by accident, Perl won't reprocess or recompile if `/o` is used.

Now, let's consider the following situation:

```
while (...)
{
    ⋮
  $regex = &GetReply('Item to find');

  foreach $item (@items) {
      if ($item =~ m/$regex/o) { # /o used for efficiency, but has a gotcha!
            ⋮
      }
  }
    ⋮
}
```

The first time through the inner `foreach` loop (of the first time through the outer `while` loop), the regular expression is processed and compiled, the result of which is then used for the actual match attempt. Because the `/o` modifier is used, the match operator in question uses the same compiled form for all its subsequent attempts. Later, in the second iteration of the outer loop, a new `$regex` is read from the user with the intention of using it for a new search. It won't work—the `/o` modifier means to compile a regex operator's regex just *once*, and since it had already been done, the original regex continues to be used—the new value of `$regex` is completely ignored.

The easiest way to solve this problem is to remove the `/o` modifier. This allows the program to work, but it is not necessarily the best solution. Even though the intermediate optimization stops the full recompile (except when the regex really has changed, the first time through each inner loop), the pre-compile processing and the check to see if it's the same as the previous regex must still be done each and every time. The resulting inefficiency is a major drawback that we'd like to avoid if at all possible.

Using the default regex to avoid work

If you can somehow ensure a successful sample match to install a regex as the default, you can reuse that match with the empty-regex construct `m//` (☞ 248):

```
while (...)
{
    ⋮
    $regex = &GetReply('Item to find');

    #  install regex (must be successful to install as default)
    if ($Sample_text !~ m/$regex/) {
        die "internal error: sample text didn't match!";
    }

    foreach $item (@items) {
       if ($item =~ m//)  #  use default regex
       {
        ⋮
       }
    }
    ⋮
}
```

Unfortunately, it's usually quite difficult to find something appropriate for the sample string if you don't know the regex beforehand. Remember, a *successful* match is required to install the regex as the default. Additionally (in modern versions of Perl), that match must not be within a dynamic scope that has already exited.

The `/o` modifier and eval

There is a solution that overcomes all of these problems. Although complex, the efficiency gains often justify the means. Consider:

```
while (...)
{
    ⋮
    $regex = &GetReply('Item to find');
    eval 'foreach $item (@items) {
                  if ($item =~ m/$regex/o) {
                                    ⋮
                  }
         }';

    # if $@ is defined, the eval had an error.
    if ($@) {
      . . .report error from eval had there been one . . .
    }
    ⋮
}
```

Notice that the entire `foreach` loop is *within* a singlequoted string, which itself is the argument to `eval`. Each time the string is `eval`uated, it is taken as a new Perl snippet, so Perl parses it from scratch, then executes it. It is executed "in place." so it has access to all the program's variables just as if it had been part of the regular code. The whole idea of using `eval` in this way is to delay the parsing until we know what each regex is to be.

What is interesting for us is that, because the snippet is parsed afresh with each `eval`, any regex operands are parsed from scratch, starting with Phase A of Figure 7-1 (☞ 223). As a consequence, the regular expression will be compiled when first encountered in the snippet (during the first time through the `foreach` loop), but not recompiled further due to the `/o` modifier. Once the `eval` has finished, that incarnation of the snippet is gone forever. The next time through the outer `while` loop, the *string* handed to `eval` is the same, but because the `eval` interprets it afresh, the *snippet* is considered new all over again. Thus, the regular expression is again new, and so it is compiled (with the new value of `$regex`) the first time it is encountered within the new `eval`.

Of course, it takes extra effort for `eval` to compile the snippet with each iteration of the outer loop. Does the `/o` savings justify the extra time? If the array of `@items` is short, probably not. If long, probably so. A few benchmarks (addressed later) can often help you decide.

This example takes advantage of the fact that, when we build a program snippet in a string to feed to `eval`, Perl doesn't consider it to be Perl code until `eval` is actually executed. You can, however, have `eval` work with a normally compiled *block* of code, instead.

Evaluating a string vs. evaluating a block

`eval` is special in that its argument can be a general expression (such as the singlequoted string just used) or a `{…}` block of code. When using the block method, such as with

```
eval {foreach $item (@items) {
             if ($item =~ m/$regex/o)
                          ⋮
             }
}};
```

the snippet is checked and compiled only once, at program load time. The intent is ostensibly to be more efficient than the one earlier (in not recompiling with each use), but here it defeats the whole point of using `eval` in the first place. We rely on the snippet being recompiled with each use, so we must use the nonblock version.

Included among the variety of reasons to use `eval` are the recompile-effects of the non-block style and the ability to trap errors. Run-time errors can be trapped with either the string or the block style, while only the string style can trap compile-time errors as well. (We saw an example of this with `$*` on page 235.) Trapping run-time errors (such as to test if a feature is supported in your version of Perl), and to trap `warn`, `die`, `exit`, and the like, are about the only reasons I can think of to use the **`eval {…}`** block version.

A third reason to use `eval` is to execute code that you build on the fly. The following snippet shows a common trick:

```
    sub Build_MatchMany_Function
    {
       my @R = @_;         # Arguments are regexes
       my $program = '';   # We'll build up a snippet in this variable
       foreach $regex (@R) {
           $program .= "return 1 if m/$regex/;"; # Create a check for
each regex
       }
       my $sub = eval "sub ( $program; return 0 }"; # create
anonymous function
       die $@ if $@;
       $sub; # return function to user
    }
```

Before explaining the details, let me show an example of how it's used. Given an array of regular expressions, `@regexes2check`, you might use a snippet like the one at the top of the next page to check lines of input.

```
# Create a function to check a bunch of regexes
$CheckFunc = Build_MatchMany_Function(@regexes2check);

while (<>)
    # Call thefunction to check the current $_
    if (&$CheckFunc) {
        ...have a line which matches one of the regexes...
    }
}
```

Given a list of regular expressions (or, more specifically, a list of strings intended to be taken as regular expressions), `Build_MatchMany_Function` builds and returns a function that, when called, indicates whether any of the regexes match the contents of `$_`.

The reason to use something like this is efficiency. If you knew what the regexes were when writing the script, all this would be unnecessary. Not knowing, you might be able to get away with

```
$regex = join('|', @regexes2check);  # Build monster regex

while (<>) {
    if (m/$regex/o) {
       ...have a line which matches one of the regexes...
    }
}
```

but the alternation makes it inefficient. (Also, it breaks if any but the first regex have backreferences.) You could also just loop through the regexes, applying them as in:

```
while (<>)
    foreach $regex (@regexes2check) {
        if (m/$regex/) {
           ...have a line which matches one of the
regexes...
           last;
        }
    }
}
```

This, too, is inefficient because each regex must be reprocessed *and* recompiled each time. Extremely inefficient. So, spending time to build an efficient match approach in the beginning can, in the long run, save a lot.

If the strings passed to `Build_MatchMany_Function` are `this`, `that`, and `other`, the snippet that it builds and `eval`uates is effectively:

```
sub {
  return 1 if m/this/;
  return 1 if m/that/;
  return 1 if m/other/;
  return 0
}
```

Each time this anonymous function is called, it checks the `$_` at the time for the three regexes, returning true the moment one is found.

It's a nice idea, but there are problems with how it's commonly implemented (including the one on the previous page). Pass `Build_MatchMany_Function` a string which contains a `$` or `@` that can be interpreted, within the `eval`, by variable interpolation, and you'll get a big surprise. A partial solution is to use a singlequote delimiter:

```
$program .= "return 1 if m'$regex';"; # Create a check for each regex
```

But there's a bigger problem. What if one of the regexes contains a singlequote (or one of whatever the regex delimiter is)? Wanting the regex ⌈`don't`⌋ adds

```
return 1 if m'don't';
```

to the snippet, which results in a syntax error when `eval`uated. You can use `\xff` or some other unlikely character as the delimiter, but why take a chance? Here's my solution to take care of these problems:

```
sub Build_MatchMany_Function
{
    my @R = @_;
    my $expr = join '||', map { "m/\$R[$_]/o" } (0..$#R);
    my $sub = eval "sub { $expr }"; # create anonymous function
    die $@ if $@;
    $sub; # return function to user
}
```

I'll leave the analysis as an exercise. However, one question: What happens if this function uses `local` instead of `my` for the `@R` array? ❖. Turn the page to check your answer.

Unsociable $& and Friends

`` $` ``, `$&`, and `$'` refer to the text leading the match, the text matched, and the text that trails the match, respectively (☞e 217). Even if the target string is later changed, these variables must still refer to the original text, as advertised. Case in point: the target string is changed immediately during a substitution, but we still need to have `$&` refer to the original (and now-replaced) text. Furthermore, even if we change the target string ourselves, `$1`, `$&`, and friends must all continue to refer to the original text (at least until the next successful match, or until the block ends).

So, how does Perl conjure up the original text despite possible changes? It makes a copy. All the variables described above actually refer to this internal-use-only copy, rather than to the original string. Having a copy means, obviously, that there are two copies of the string in memory at once. If the target string is huge, so is the duplication. But then, since you need the copy to support these variables, there is really no other choice, right?

Internal optimizations

Not exactly. If you don't intend to use these special variables, the copy is obviously unnecessary, and omitting it can yield a huge savings. The problem is that Perl doesn't realize that you have no intention to use the variables. Still, Perl

`local` vs. `my`

❖ Answer to the question on page 273.

Before answering the question, first a short summary of *binding*: Whenever Perl compiles a snippet, whether during program load or `eval`, references to variables in the snippet are "linked" (*bound*) to the code. The *values* of the variables are not accessed during program load time, the variables don't have values yet. Values are accessed when the code is actually executed.

`my @R` creates a new variable, private, distinct, and unrelated to all other variables in the program. When the snippet to create the anonymous subroutine (the *a-sub*) is `eval`uated, its code is linked to our private `@R`. "`@R`" refer to this private variable. They don't *access* `@R` yet, since that happens only when the a-sub is executed.

When `Build_MatchMany_Function` exits, the private `@R` would normally disappear, but since the a-sub still links to it, `@R` and its strings are kept around, although they become inaccessible to all but the a-sub (references to "`@R`" elsewhere in the code refer to the unrelated global variable of the same name). When the a-sub is later executed, that private `@R` is referenced and our strings are used as desired.

`local @R`, on the other hand, simply saves a copy of the global variable `@R` before we overwrite it with the `@_` array. It's likely that there was no previous value, but if some other part of the program happens to use it (the global variable `@R`, that is) for whatever reason, we've saved a copy so we don't interfere, just in case. When the snippet is evaluated, the "`@R`" links to the same global variable `@R`. (This linking is unrelated to our use of `local`—since there is no private `my` version of `@R`, references to "`@R`" are to the global variable, and any use or non-use of `local` to copy data is irrelevant.)

When `Build_MatchMany_Function` exits, the `@R` copy is restored. The global variabe `@R` is the same variable as it was during the `eval`, and is still the same variable that the a-sub links to, but the *content* of the variable is now different. (We'd copied new values into `@R`, but *never* referenced them! A completely wasted effort.) The a-sub expects the global variable `@R` to hold strings to be used as regular expressions, but we've already lost the values we'd copied into it. When the subroutine is first used, it sees what `@R` happens to have at the time whatever it is, it's not our regexes, so the whole approach breaks down.

Mmm. If before we leave `Build_MatchMany_Function`, we actually use the a-sub (and make sure that the sample text cannot match any of the regexes), the `/o` would lock in our regexes while `@R` still holds them, getting around the problem. (If the sample text matches a regex, only the regexes to that point are locked in). This might actually be an appealing solution—we need `@R` only until all the regexes are locked in. After that, the strings used to create them are unneeded, so keeping them around in the separate (but undeleteable until the a-sub is deleted) `my @R` wastes memory.

The /e Modifier

Only the substitution operator allows the use of `/e` modifier. When used, the replacement operand is evaluated as if with **`eval {`**…**`}`** (including the load-time syntax check), the result of which is substituted for the matched text. The replacement operand does not undergo any processing before the `eval` (except to determine its lexical extent, as outlined in Figure 7-1's Phase A), not even singlequotish processing.*[17☞308]* The actual evaluation, however, is redone upon each match.

As an example, you can encode special characters of a World Wide Web `URL` using `%` followed by their two-digit hexadecimal representation. To encode all non-alphanumerics this way, you can use

```
    $url =~ s/([^a-zA-Z0-9])/sprintf('%%%02x', ord($1))/ge;
```

and to decode it back, you can use:

```
    $url =~ s/%([0-9a-f][0-9a-f])/pack("C",hex($1))/ige;
```

In short, **`pack("C",`** ***value*****`)`** converts from a numeric value to the character with that value, while **`sprintf('%%%02x', ord(`*****character*****`))`** does the opposite; see your favorite Perl documentation for more information. (Also, see the footnote on page 66 for more on this example.)

The moving target of interpretation

Particularly with the `/e` modifier, you should understand exactly *who* interprets *what* and *when.* It's not too confusing, but it does take some effort to keep things straight. For example, even with something as simple as **`s/…/'echo $$'/e`**, the question arises whether it's Perl or the shell that interprets the `$$`. To Perl and many shells, `$$` is the process `ID` (of Perl or the shell, as the case may be). You must consider several levels of interpretation. First, the replacement-operand has no `pre-eval` processing in Perl5, but in Perl4 has singlequotish processing. When the result is `evaluated`, the backquotes provide doublequoted-string processing. (This is when Perl interpolates `$$` it may be escaped to prevent this interpolation.) Finally, the result is sent to the shell, which then runs the *echo* command. (If the `$$` had been escaped, it would be passed to the shell unescaped, resulting in the shell's interpolation of `$$`.)

To add to the fray, if using the `/g` modifier as well, should `'echo $$'` be evaluated just once (with the result being used for all replacements), or should it be done after each match? When `$1` and such appear in the replacement operand, it obviously must be evaluated on a per-match basis so that the `$1` properly reflects its after-match status. Other situations are less clear. With this `echo` example, Perl version 5.000 does only one evaluation, while other versions before and after evaluate on a per-match basis.

/eieio

Perhaps useful only in an Obfuscated Perl Contest, it is interesting to note that the replacement operand will be `eval`uated multiple times if `/e` is specified more than once. (It is the only modifier for which repetition matters.) This is what Larry 'Wall calls an *accidental feature,* and was "discovered" in early 1991. During the ensuing `comp.lang.perl` discussion, Randal Schwartz offered one of his patented JAPH signatures:*

```
   $Old_MacDonald = q#print #; $had_a_farm = (q-q:Just another
Perl hacker,:-);
   s/^/q[Sing it, boys and
girls…],$Old_MacDonald.$had_a_farm/eieio;
```

The `eval` due to the first `/e` sees

```
    q[Sing it, boys and girls…],$Old_MacDonald.$had_a_farm
```

whose execution results in `print q:Just another Perl hacker,:` which then prints Randal's "Just another Perl hacker" signature when evaluated due to the second `/e`.

Actually, this kind of construct is sometimes useful. Consider wanting to interpolate variables into a string manually (such as if the string is read from a configuration file). A simple approach uses **`$data =~ s/(\$[a-zA-Z_]\w*)/$1/eeg;`**. Applying this to '`option=$var`', the regex matches `option=$var`. The first `eval` simply sees the snippet `$1` as provided in the replacement-operand, which in this case expands to `$var`. Due to the second `/e`, this result is evaluated again, resulting in whatever value the variable `$var` has at the time. This then replaces the matched '`$var`', in effect interpolating the variable.

I actually use something like this with my personal Web pages most of them are written in a pseudo Perl/HTML code that gets run through a CGI when pulled by a remote client. It allows me to calculate things on the fly, such as to remind readers how few shopping days are left until my birthday.**

Context and Return Value

Recall that the match operator returns different values based upon the particular combination of context and `/g`. The substitution operator, however, has none of these complexities—it returns the same type of information regardless of either concern.

The return value is either the number of substitutions performed or, if none were done, an empty string.*[18☞308]* When interpreted as a Boolean (such as for the

---

[*] My thanks to Hans Mulder for providing the historical background for this section, and for Randal for being Just Another Perl Hacker with a sense of humor.

[**] If you, too, would like to see how many days are left until my birthday, just load
`http://omrongw.wg.omron.co.jp/cgi-bin/j-e/jfriedl.html`
or perhaps one of its mirrors (see Appendix A).

conditional of an `if`), the return value conveniently interprets as true if any substitutions were done, false if not.

Using /g with a Regex That Can Match Nothingness

The earlier section on the match operator presented a detailed look at the special concerns when using a regex that can match nothingness. Different versions of Perl act differently, which muddies the water considerably. Fortunately, all versions' substitution works the same in this respect. Everything presented earlier in Table 7-9 (☞ 250) applies to how `s/…/…/g` matches as well, but the entries marked "Perl5" are for the match operator only. The entries marked "Perl4" apply to Perl4's match operator, and all versions' substitution operator.

The Split Operator

The multifaceted `split` operator (often called a *function* in casual conversation) is commonly used as the converse of a list-context `m/…/g` (☞ 253). The latter returns text matched by the regex, while a `split` with the same regex returns text *separated* by matches. The normal match **`$text =~ m/:/g`** applied against a `$text` of '`IO.SYS:225558:95-10-03:-a-sh:optional`', returns the four-element list

```
(':', ':', ':', ':')
```

which doesn't seem useful. On the other hand, **`split(/:/, $text)`** returns the five-element list:

```
('IO.SYS',  '225558',  '95-10-03',  '-a-sh',  'optional')
```

Both examples reflect that ⌈`:`⌋ matches four times. With `split`, those four matches partition a copy of the target into five chunks which are returned as a list of five strings.

In its most simple form with simple data like this, `split` is as easy to understand as it is useful. However, when the use of `split` or the data are complicated, understanding is less clear-cut. First, I'll quickly cover some of the basics.

Basic Split

split is an operator that looks like a function, and takes up to three operands:

```
split(match operand, target string, chunk-limit operand)
```

(The parentheses are optional with Perl5.) Default values, discussed below, are provided for operands left off the end.

Basic match operand

The match operand has several special-case situations, but normally it's is a simple regular expression match such as `/:/` or `m/\s*<P>\s*/i`. Conventionally, …

sometimes realizes that it doesn't need to make the copy. If you can get your code to trigger such cases, it will run more efficiently. Eliminating the copy not only saves time, but as a surprise mystery bonus, the substitution operator is often more efficient when no copy is done. (This is taken up later in this section.)

Warning: The situations and tricks I describe exploit *internal* workings of Perl. It's nice if they make your programs faster, but they're not part of the Perl specification* and *may be changed in future releases.* (I am writing as of version 5.003.) If these optimizations suddenly disappear, the only effect will be on efficiency programs will still produce the same results, so you don't need to worry that much.

Three situations trigger the copy for a successful match or substitution *[23 ☞ 308]:*

- the use of `` $` ``, $&, or `$'` anywhere in the *entire* script
- the use of capturing parentheses in the regex
- the use of the `/i` modifier with a non `-/g` match operator

Also, you might need *additional* internal copies to support:

- the use of the `/i` modifier (with any match or substitute)
- the use of many, but not all, substitution operators

I'll go over the first three here, and the other two in the following section.

`` $` ``, $&, or `$'` require the copy

Perl must make a copy to support any use of `$``, $&, and `$'`. In practice, these variables are not used after most matches, so it would be nice if the copy were done only for those matches that needed it. But because of the dynamically scoped nature of these variables, their use may well be some distance away from the actual match. Theoretically, it may be possible for Perl to do exhaustive analysis to determine that all uses of these variables can't possibly refer to a particular match (and thus omit the copy for that match), in practice, Perl does not do this. Therefore, it must normally do the copy for every successful match of all regexes during the entire run of the program.

However, it does notice whether there are no references *whatsoever* to `$``, $&, or `$'` in the entire program (including all libraries referenced by the script!). Since the variables never appear in the program, Perl can be quite sure that a copy merely to support them can safely be omitted. Thus, if you can be sure that your code and any libraries it might reference never use `$``, $&, or `$'`, you are not penalized by the copy except when explicitly required by the two other cases.

---

* Well, they wouldn't be a part *if there were* a Perl specification.

Capturing parentheses require the copy

If you use capturing parentheses in your regex, Perl assumes that you intend to use the captured text so does the copy after the match. (This means that the eventual use or non-use of `$1` has no bearing whatsoever—if capturing parentheses are used, the copy is made even if its results are never used.) In Perl4 there were no grouping-only parentheses, so even if you didn't intend to capture text, you did anyway as a side effect and were penalized accordingly. Now, with ⌈`(?:…)`⌋, you should never find that you are capturing text that you don't intend to use.* But when you do intend to use `$1`, `$2`, etc., Perl will have made the copy for you.

`m/…/i` requires the copy

A non-`/g` match operator with `/i` causes a copy. Why? Frankly, I don't know. From looking at the code, the copy seems entirely superfluous to me, but I'm certainly no expert in Perl internals. Anyway, there's another, more important efficiency hit to be concerned about with `/i`. I'll pick up this subject again in a moment, but first I'd like to show some benchmarks that illustrate the effects of the `$&`-support copy.

An example benchmarked

I ran a simple benchmark that checked `m/c/` against each of the 50,000 or so lines of C that make up the Perl source distribution. The check merely noted whether there was a '`c`' on a line the benchmark didn't actually do anything with the information since the goal was to determine the effect of the behind-the-scenes copying. I ran the test two different ways: once where I made sure not to trigger any of the conditions mentioned above, and once where I made sure to do so. The only difference, therefore, was in the extra copy overhead.

The run with the extra copying consistently took over 35 percent longer than the one without. This represents an "average worst case," so to speak. The more real work a program does, the less of an effect, percentage-wise, the copying has. The benchmark didn't do any real work, so the effect is highlighted.

On the other hand, in true worst-case scenarios, the extra copy might truly be an overwhelming portion of the work. I ran the same test on the same data, but this time as *one huge line* incorporating the more than megabyte of data rather than the 50,000 or so reasonably sized lines. Thus, the relative performance of a single match can be checked. The match without the copy returned almost immediately, since it was sure to find a '`c`' somewhere near the start of the string. Once it did, it was done. The test with the copy is the same except, well, it had to make a copy of the megabyte-plus-sized string first. Relatively speaking, it took over 700 times longer! Knowing the ramifications, therefore, of certain constructs allows you to tweak your code for better efficiently.

---

* Well, capturing parentheses are also used for backreferences, so it's possible that capturing parentheses might be used when `$1` and the like are not. This seems uncommon in practice.

Conclusions and recommendations about $& and friends

It would be nice if Perl knew the programmer's intentions and made the copy only as necessary. But remember, the copies are not "bad"   Perl's handling of these bookkeeping drudgeries behind the scenes is why we use it and not, say, C or assembly language. Indeed, Perl was first developed in part to free users from the mechanics of bit-fiddling so they could concentrate on creating solutions to problems.

A solution in Perl can be approached in many ways, and I've said numerous times that if you write in Perl as you write in another language (such as C), your Perl will be lacking and almost certainly inefficient. For the most part, crafting programs *The Perl Way* should go a long way toward putting you on the right track, but still, as with any discipline, special care can produce better results. So yes, while the copies aren't "wrong," we still want to avoid unnecessary copying whenever possible. Towards that end, there are steps we can take.

Foremost, of course, is to never use `` $` ``, $&, or `$'`, *anywhere* in your code. This also means to never use `English.pm` nor any library modules that use it, or in any other way references these variables. Table 7-10 shows a list of standard libraries (in Perl version 5.003) which reference one of the naughty variables, or uses another library that does. You'll notice that most are tainted only because they use `Carp.pm`. If you look into that file, you'll find only one naughty variable:

```
    $eval =~ s/[\\\']/\\$&/g;
```

Changing this to

```
    $eval =~ s/([\\\'])/\\$1/g;
```

makes most of the standard libraries sociable in this respect. Why hasn't this been done to the standard distribution? I have no idea. Hopefully, it will be changed in a future release.

If you can be sure these variables never appear, you'll know you will do the copy only when you explicitly request it with capturing parentheses, or via the rogue `m/…/i`. Some expressions in current code might need to be rewritten. On a case-by-case basis, `$`` ` can often be mimicked by ⌈`(.*?)`⌋ at the head of the regex, `$&` by ⌈`(…)`⌋ around the regex, and `$'` by ⌈`(?=(.*))`⌋ at the end of the regex.

If your needs allow, there are other, non-regex methods that might be attempted in place of some regexes. You can use `index(…)` to find a fixed string, for example. In the benchmarks I described earlier, it was almost 20 percent faster than `m/…/`, even without the copy overhead.

Table 7-10: Standard Libraries That Are Naughty (That Reference $& and Friends)

C	AutoLoader	C	Fcntl	+C	Pod::Text
C	AutoSplit		File::Basename	C	POSIX
C	Benchmark	C	File::Copy	C	Safe
C	Carp	C	File::Find	C	SDBM File
C	DB_File	C	File::Path	C	SelectSaver
+CE	diagnostics	C	FileCache	C	SelfLoader
C	DirHandle	C	FileHandle	C	Shell
	dotsh.pl	C	GDBM File	C	Socket
	dumpvar.pl		Getopt::Long	C	Sys::Hostname
C	DynaLoader	C	IPC::Open2	C	Syslog
	English	+C	IPC::Open3	C	Term::Cap
+CB	ExtUtils:: Install	C	lib	C	Test:: Harness
+CB	ExtUtils::Liblist	C	Math::BigFloat	C	Text::ParseWords
C	ExtUtils::MakeMaker	C	MMVMS	C	Text::Wrap
+CB	ExtUtils::Manifest	+CL	newgetopt.pl	C	Tie::Hash
C	ExtUtils::Mkbootstrap	C	ODBMFile	C	Tie::Scalar
C	ExtUtils::Mksymlists		open2.pl	C	Tie::SubstrHash
+CB	ExtUtils::MMUnix		open3.pl	C	Time::Local
C	ExtUtils::testlib		perl5db.pl	C	vars

Naughty due to the use of: C: `Carp` B: `File::Basename` E: `English` L: `Getopt::Long`

The Efficiency Penalty of the `/i` Modifier

If you ask Perl do match in a case-insensitive manner, common sense tells you that you are asking for more work to be done. You might be surprised, though, to find out just how much extra work that really is.

Before a match or substitution operator applies an `/i`-governed regex, Perl first makes a temporary copy of the *entire* target string. This copy is in addition to any copy in support of $& and friends. The latter is done only after a successful match, while the one to support a case-insensitive match is done before the attempt. After the copy is made, the engine then makes a *second* pass over the entire string, converting any uppercase characters to lowercase. The result might happen to be the same as the original, but in any case, all letters are lowercase.

This goes hand in hand with a bit of extra work done during the compilation of the regex to an internal form. At that time, uppercase letters in the regex are converted to lowercase as well.

The result of these two steps is a string and a regex that then matches normally—nothing special or extra needs to be done within the actual matching portion of the regex engine. It all appears to be a very tidy arrangement, but this has got to be one of the most gratuitous inefficiencies in all of Perl.

Methods to implement case-insensitive matching

There are (at least) two schools of thought on how to implement case-insensitive matching. We've just seen one, which I call *string oriented* (in addition to "gratuitously inefficient"). The other, which I consider to be far superior, is what I would call *regex oriented*. It works with the original mixed-case target string, having the engine itself make allowances for case-insensitive matching as the need arises.

How to check whether your code is tainted by `$&`

Especially with the use of libraries, it's not always easy to notice whether your program ever references `$&`, `` $` ``, or `$'`. I went to the trouble of modifying my version of Perl to issue a warning the first time one was encountered. (If you'd like to do the same, search for the three appearances of `sawampersand` in Perl's *gv.c*, and add an appropriate call to warn.)

An easier approach is to test for the performance penalty, although it doesn't tell you where the offending variable is. Here's a subroutine that I've come up with:

```
sub CheckNaughtiness
{
  local($_) = 'x' x 10000; # some non-small amount of data

  # calculate the overhead of a do-nothing loop
  local($start) = (times)[0];
  for ($i = 0; $i < 5000; $i++)              {       }
  local($overhead) = (times)[0] - $start;

  # now calculate the time for the same number
  $start = (times)[0];
  for ($i = 0; $i < 5000; $i++)                   { m/^/; }
  local($delta) = (times)[0] - $start;

  # a differential of 10 is just a heuristic
  printf "It seems your code is %s (overhead=%.2f, delta=%.2f)\n",
    ($delta > $overhead*10) ? "naughty":"clean", $overhead, $delta;
}
```

This is not a function you would keep in production code, but one you might insert temporarily and call once at the beginning of the program (perhaps immediately following it with an `exit`, then removing it altogether once you have your answer). Once you know you program is `$&-clean`, there is still a chance that a rogue `eval` could introduce it during the run of the program, so it's also a good idea to test at the end of execution, just to be sure.

Many subexpressions (and full regular expressions, for that matter) do not require special handling: the CSV program at the start of the chapter (☞ 205), the regex to add commas to numbers **(**☞ 229), and even the huge, 4,724-byte regex we construct in "Matching an Email Address" (☞ 294) are all free of the need for special case-insensitive handling. A case-insensitive match with such expressions **should not** have any efficiency penalty at all.

Even a character class with letters shouldn't entail an efficiency penalty. At compile time, the appropriate other-case version of any letter can easily be included. (A character class's efficiency is not related to the number of characters in the class; ☞ 115.) So, the only real extra work would be when letters are included in literal text, and with backreferences. Although they must be dealt with, they can certainly be handled more efficiently than making a copy of the entire target string.

By the way, I forgot to mention that when the `/g` modifier is used, the copy is done with *each* match. At least the copy is only from the start of the match to the end of the string with a `m/…/ig` on a long string, the copies are successively shorter as the matches near the end.

A few `/i` benchmarks

I did a few benchmarks, similar to the ones on page 276. As before, the test data is a 52,011-line, 1,192,395-byte file made up of the Perl's main C source.

As an unfair and cruel first test, I loaded the entire file into a single string, and benchmarked **`1 while m/./g`** and **`1 while m/./gi`**. Dot certainly doesn't care one way or the other about capitalization, so it's not reasonable to penalize this match for case-insensitive handling. On my machine, the first snippet benchmarked at a shade under 12 seconds. Simply adding the `/i` modifier (which, you'll note, is meaningless in this case) slowed the program by four *orders of magnitude*, to over a day and a half!* I calculate that the needless copying caused Perl to shuffle around more than 647,585 *megabytes* inside my CPU. This is particularly unfortunate, since it's so trivial for the compilation part of the engine to tell the matching part that case-insensitiveness is irrelevant for ⌈`.`⌋, the regex at hand.

This unrealistic benchmark is definitely a worst-case scenario. Searching a huge string for something that matches less often than ⌈`.`⌋ is more realistic, so I benchmarked **`m/\bwhile\b/gi`** and **`m/\b[wW][hH][iI][lL][eE]\b/g`** on the same string. Here, I try to mimic the regex-oriented approach myself. It's incredibly naive for a regex-oriented implementation to actually turn literal text into character classes,** so we can consider the `/i`-equivalent to be a worst-case situation in this respect. In fact, manually turning ⌈`while`⌋ into ⌈`[wW][hH][iI][lL][eE]`⌋ also kills Perl's fixed string check (☞ 155), and renders `study(`☞ 287) useless for the regex. With all this against it, we should expect it to be very slow indeed. But it's still over 50 times faster than the `/i` version!

Perhaps this test is still unfair   the `/i`-induced copy made at the start of the match, and after each of the 412 matches of ⌈`\bwhile\b`⌋ in my test data, is large. (Remember, the single string is over a megabyte long.) Let's try testing **`m/^int/i`** and **`m/^[iI][nN][tT]/`** on each of the 50,000 lines of the test file. In this case, `/i` has each line copied before the match attempt, but since they're so short, the copy is not so crushing a penalty as before: the `/i` version is now just 77 percent slower. Actually, this includes the extra copies inexplicably made for each of the 148 matches   remember, a non-`/g m/…/i` induces the `$&`-support copy.

---

* I didn't actually run the benchmark that long. Based on other test cases, I calculated that it would take about 36.4 hours. Feel free to try it yourself, though.

** Although this is exactly what the original implementation of *grep* did!

Final words about the `/i` penalty

As the later benchmarks illustrate, the `/i`-related penalty is not as heinous as the first benchmark leads you to believe. Still, it's a concern you should be very aware of, and I hope that future versions of Perl eliminate the most outlandish of these inefficiencies.

Foremost: don't use `/i` unless you really have to. Blindly adding it to a regex that doesn't require it invites many wasted CPU cycles. In particular, when working with long strings, it can be a *huge* benefit to rewrite a regex to mimic the regex-oriented approach to case insensitivity, as **I** did with the last two benchmarks.

Substitution Efficiency Concerns

As I mentioned, I am not an expert in Perl's internal workings. When it comes to how Perl's substitute operator actually moves strings and substrings around internally during the course of a substitute, well, I'm pretty much completely lost. The code and logic are not for the faint of heart.

Still, I have managed to understand a bit about how it works,* and have developed a few rules of thumb that I'd like to share with you. Let me warn you, up front, that there's no simple one-sentence summary to all this. Perl often takes internal optimizations in bits and pieces where they can be found, and numerous special cases and opportunities surrounding the substitution operator provide for fertile ground for optimizations. It turns out that the `$&`-support copy disables all of the substitution-related optimizations, so that's all the more reason to banish `$&` and friends from your code.

Let me start by stepping back to look at the substitution operator efficiency's worst-case scenario.

The normal "slow-mode" of the substitution operator

In the worst case, the substitution operator simply builds the new copy of the target string, off to the side, then swaps it for the original. As an example, the temperature conversion one-liner from the first page of this chapter

```
s⌊(\d+(\.\d*)?)F\b⌋{sprintf "%.0fC", ($1-32) * 5/9}eg
```

matches at the marked locations of:

```
Water boils at 212F, freezes at 32F.
```

When the first match is found, Perl creates an empty temporary string and then copies everything before the match, '`Water•boils•at.`', to it. The substitution text is then computed ('`100C`' in this case) and added to the end of the temporary string. (By the way, it's at this point that the `$&`-support copy would be made were it required.)

---

* I modified my copy of version 5.003 to spit out volumes of pretty color-coded messages as various things happen internally. This way, I have been able to understand the overall picture without having to understand the fine details.

At the next match (because `/g` is used), the text between the two matches is added to the temporary string, followed by the newly computed substitution text, '`0C`'. Finally, after it can't find any more matches, the remainder of the string (just the final period in this case) is copied to close out the temporary string. This leaves us with:

```
Water boils at 100C, freezes at 0C.
```

The original target string, `$_`, is then discarded and replaced by the temporary string. (I'd think the original target could be used to support `$1`, `$&`, and friends, but it does not   a separate copy is made for that, if required.)

At first, this method of building up the result might seem reasonable because for the general case, it *is* reasonable. But imagine something simple like `s/\s+$//` to remove trailing whitespace. Do you really need to copy the whole (potentially huge) string, just to lop off its end? In theory, you don't. In practice, Perl doesn't either. Well, at least not always.

$& and friends disable all substitute-operator optimizations

Perl is smart enough to optimize `s/\s+$//` to simply adjusting the length of the target string. This means no extra copies   very fast. For reasons that escape me, however, this optimization (and all the substitution optimizations I mention in a moment) are disabled when the `$&`-support copy is done. Why? I don't know, but the practical effect is yet another way that `$&` is detrimental to your code's efficiency.

The `$&`-support copy is also done when there are capturing parentheses in the regex, although in that case, you'll likely be enjoying the fruits of that copy (since `$1` and the like are supported by it). Capturing parentheses also disable the substitution optimizations, but at least only for the regexes they're used in and not for all regexes as a single stray `$&` does.

Replacement text larger than the match not optimized

`s/\s+$//` is among many examples that fit a pattern: when the replacement text is shorter or the same length as the text being replaced, it can be inserted right into the string. There's no need to make a full copy. Figure 7-2 shows part of the example from page 45, applying the substitution `s/<FIRST>/Tom/` to the string '`Dear <FIRST>,NL`'. The new text is copied directly over what is being replaced, and the text that follows the match is moved down to fill the gap.

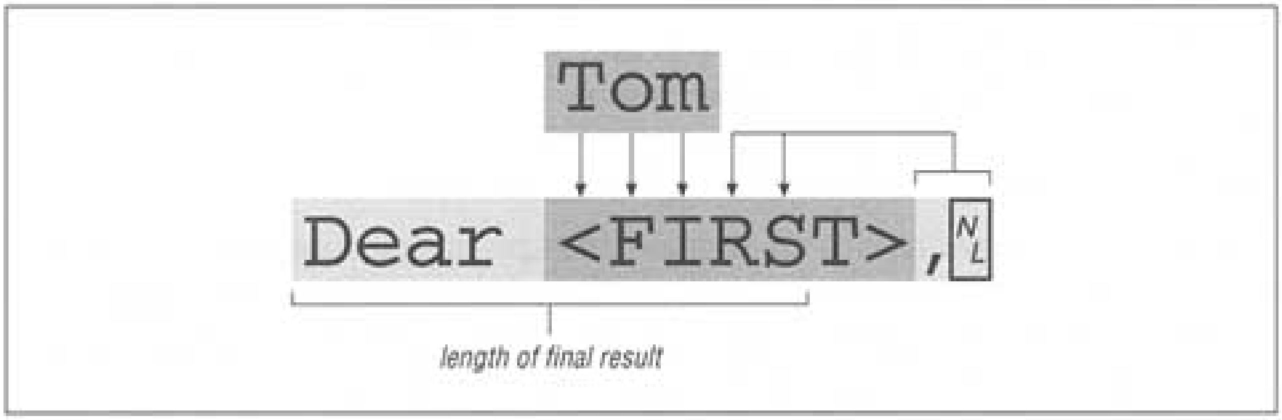


Figure 7-2:
Applying `s/<FIRST>/Tom/` to '`Dear•<FIRST>,`NL'

In the case of `s/\s+$//`, there's no replacement text to be filled in and no match-following text to be moved down once the match is found, the size of the string is adjusted to lop off the match, and that's that. Very zippy. The same kinds of optimizations also apply to matches at the beginning of the string.

When the replacement text is exactly the same size as the matched text, you'd think that as a further optimization, the "move to fill the gap" could be omitted, since there is no gap when the sizes are an exact match. For some reason, the needless "move" is still done. At least the algorithm, as it stands, never copies more than half the string (since it is smart enough to decide whether it should copy the part before the match, or the part after).

A substitution with `/g` is a bit better still. It doesn't do the gap-filling move until it knows just how much to move (by delaying the move until it knows where the next match is). Also, it seems in this case that the worthless move to fill a nonexistent gap is kindly omitted.

Only fixed-string replacement text substitutions are optimized

These optimizations kick in only when Perl knows the size of the replacement string *before* the overall match attempt begins. This means that any replacement string with `$1` and friends disables the optimizations.* Other variable interpolation, however, does not. With the previous example, the original substitution was:

```
$given = 'Tom';
$letter =~ s/<FIRST>/$given/g;
```

The replacement operand has variables interpolated *before* any matching begins, so the size of the result is known.

---

[*] It is redundant to say that the use of `$1` in the replacement string disables the optimizations. Recall that the use of capturing parentheses in the regex causes the `$&`-support copy, and that copy also disables the substitution optimizations. It's silly to use `$1` without capturing parentheses, as you're guaranteed its value will be undefined (☞ 217).

Any substitution using the `/e` modifier, of course, doesn't know the size of the substitution text until *after* a match, and the substitution operand is `eval`uated, so there are no substitution optimizations with `/e` either.

Final comments on all these internal optimizations

When it comes down to it, with all these attempts at optimizations, the infamous YMMV (your mileage may vary) applies. There are a bazillion more details and special cases than I've presented here, and it's certain that some internal workings will change in future versions. If it is important enough to want to optimize, it is, perhaps, important enough to benchmark.

Benchmarking

If you really care about efficiency, it may be best to try benchmarking. Perl5 provides the `Benchmark` module, but it is tainted by the `$&` penalty. This is quite unfortunate, for the penalty might itself silently render the benchmark results invalid. I prefer to keep things simple by just wrapping the code to be tested in something like:

```
$start = (times)[0];
    ⋮
$delta = (times)[0] - $start;
printf "took %.lf seconds\n", $delta;
```

An important consideration about benchmarking is that due to clock granularity (1/60 or 1/100 of a second on many systems), it's best to have code that runs for at least a few seconds. If the code executes too quickly, do it over and over again in a loop. Also, try to remove unrelated processing from the timed portion. For example, rather than

```
$start = (times)[0]; #  Go! Start the clock.
   $count = 0;
   while (<>)
      $count++ while m/\b(?:char\b|return\b|void\b)/g;
   }
   print "found $count items.\n";
$delta = (times)[0] - $start; #  Done. Stop the clock.
printf "the benchmark took %.lf seconds.\n", $delta;
```

it is better to do:

```
    $count = 0;          #  (no need to have this timed, so bring above the clock
start)
    @lines = <>;         #  Do all file I/O here, so the slow disk is not an issue
when timed
    $start = (times)[0]; #  Okay, I/O now done, so now start the clock.
      foreach (@lines) {
         $count++ while m/\b(?:char\b|return\b|void\b)/g;
      }
    $delta = (times)[0] - $start;  #  Done. Stop the clock.
    print "found $count items.\n"; #  (no need to have this timed)
    printf "the benchmark took %.lf seconds.\n", $delta;
```

The biggest change is that the file I/O has been moved out from the timed portion. Of course, if you don't have enough free memory and start swapping to disk, the advantage is gone, so make sure that doesn't happen. You can simulate more data by using a smaller amount of real data and processing it several times:

```
for ($i = 0; $i < 10; $i++) {
   foreach (@lines) {
        $count++ while m/\b(?:char\b|return\b|void\b)/g;
   }
}
```

It might take some time to get used to benchmarking in a reasonable way, but the results can be quite enlightening and downright rewarding.

Regex Debugging Information

Perl carries out a phenomenal number of optimizations to try to arrive at a regex match result quickly; some of the less esoteric ones are listed in Chapter 5's "Internal Optimizations" ☞ 154). If your *perl* has been compiled with debugging information (by using `-DDEBUGGING` during its build), the `-D` debugging command-line option is available. The use of `-Dr` (`-D512` with Perl4) tells you a bit about how Perl compiles your regular expressions and gives you a blow-by-blow account of each application.

Much of what `-Dr` provides is beyond the scope of this book, but you can readily understand some of its information. Let's look at a simple example (I'm using Perl version 5.003):

```
1      jfriedl@tubby> perl -cwDr -e '/^Subject: (.*)/'
2      rarest char j at 3
3      first 14 next 83 offset 4
4       1:BRANCH(47)
5       5:BOL(9)
6       9:EXACTLY(23) <Subject: >
7      23:OPEN1(29)
          ⋮
8      47:END(0)
9      start 'Subject: ' anchored minlen 9
```

At 1, I invoke *perl* at my shell prompt, using the command-line arguments `-c` (which means check script, don't actually execute it), `-w` (issue warnings about things Perl thinks are dubious always used as a matter of principle), `-Dr` (regex debugging), and `-e` (the next argument is the Perl snippet itself). This combination is convenient for checking regexes right from the command line. The regex here is ⌈`^ Subject:•(.*)`⌋ which we've seen several times in this book.

Lines 4 through 8 represents Perl's compiled form of the regex. For the most part, we won't be concerned much about it here. However, in even a casual look, line 6 sticks out as understandable.

Literal text cognizance

Many of Perl's optimizations are contingent upon it deducing from the regex some fixed literal text which must appear in any possible match. (I'll call this "literal text cognizance.") In the example, that text is '`Subject:•`', but many expressions either have no required literal text, or have it beyond Perl's ability to deduce. (This is one area where Emacs' optimization far outshines Perl's; ☞ 197.) Some examples where Perl can deduce nothing include ⌜`-?([0-9]+(\.[0-9]*)?|\.[0-9]+)`⌟, ⌜`^\s*`⌟, ⌜`^(-?\d+)(\d{3})`⌟, and even ⌜`int|void|while`⌟.

Examining ⌜`int|void|while`⌟, you see that '`i`' is required in any match. Some NFA engines can deduce exactly that (any DFA knows it implicitly), but Perl's engine is unfortunately not one of them. In the debugging output, `int`, `void`, and `while` appear on lines similar to 6 above, but those are local (subexpression) requirements. For literal text cognizance, Perl needs global regex-wide confidence, and as a general rule, it can't deduce fixed text from anything that's part of alternation.

Many expressions, such as ⌜`<CODE>(.*?)</CODE>`⌟, have more than one clump of literal text. In these cases, Perl (somewhat magically) selects one or two of the clumps and makes them available to the rest of the optimization subroutines. The selected clump(s) are shown on a line similar to 9.

Main optimizations reported by -Dr

Line 9 can report a number of different things. Some items you might see include:

`start` *'clump'*

> Indicates that a match must begin with *clump,* one of those found by literal text cognizance. This allows Perl to do optimizations such as the *fixed string check* and *first character discrimination* discussed in Chapter 5.

`must have` *"clump"* `back` *num*

Indicates a required clump of text like the `start` item above, but the clump is not at the start of the regex. If *num* is not `-1`, Perl knows any match must begin that many characters earlier. For example, with ⌈`[Tt]ubby`…⌋, the report is '`must have "ubby" back 1`', meaning that if a fixed string check reveals `ubby` starting at such and such a location, the whole regex should be applied starting at the previous position.

Conversely, with something like ⌈`.*tubby`⌋, it's not helpful to know exactly where `tubby` might be in a string, since a match including it could start at any previous position, so *num* is `-1`.

`stclass` ':*kind*'

Indicates that Perl realizes the match must start with some particular kind of character. With ⌈`\s+`⌋, *kind* is `SPACE`, while with ⌈`\d+`⌋ it is `DIGIT`. With a character class like the ⌈`[Tt]ubby`⌋ example, *kind* is reported as `ANYOF`.

`plus`

Indicates that the `stclass` or a single-character start item is governed by ⌈`+`⌋, so not only can the *first character discrimination* find the start of potential matches, but it can also quickly zip past a leading ⌈`\s+`⌋ and the like before letting the full (but slower) regex engine attempt the complete match.

`anchored`

Indicates that the regex begins with a caret anchor. This allows the "String/Line Anchor" optimization (☞ 158).

`implicit`

Indicates that Perl has added an implicit caret to the start of the regex because the regex begins with ⌈`.*`⌋ (☞ 158).

Other optimizations arising from literal text cognizance

One other optimization arising from literal text cognizance relates to `study` (which is examined momentarily). Perl somewhat arbitrarily selects one character it considers "rare" from the selected clump(s). Before a match, if a string has been `study`ed, Perl knows immediately if that character exists anywhere in the string. If it doesn't exist, no match is possible and the regex engine does not need to get involved at all. It's a quick way to prune some impossible matches. The character selected is reported at 2.

For the trivia-minded, Perl's idea of the rarest character is `\000`, followed by `\001`, `\013`, `\177`, and `\200`. Some of the rarest printable characters are `~`, `Q`, `Z`, `?`, and `@`. The least-rare characters are `e`, space, and `t`. (The manpage says that this was derived by examining a combination of C programs and English text.)

The Study Function

As opposed to optimizing the regex itself, `study(…)` optimizes access to certain information about a *string.* A regex, or multiple regexes, can then benefit from the cached knowledge when applied to the string. What it does is simple, but understanding when it's a benefit or not can be quite difficult. It has no effect whatsoever[*] on any values or results of a program the only effects are that Perl uses more memory, and that overall execution time might increase, stay the same, or (here's the goal) decrease.

---

[*] Or, at least it shouldn't in theory. However, as of Perl version 5.003, there is a bug in which the use of `study` can cause successful matches to fail. This is discussed further at the end of this section.

When you study a string, Perl takes some time and memory to build a list of places in the string each character is found. On most systems, the memory required is four times the size of the string (but is reused with subsequent calls of `study`). `study`'s benefit can be realized with each subsequent regex match against the string, but only until the string is modified. Any modification of the string renders the `study` list invalid, as does `studying` a different string.

The regex engine itself never looks at the `study` list; only the transmission references it. The transmission looks at the `start` and `must have` debugging information mentioned on page 286 to pick what it considers a *rare character* (discussed on the previous page). It picks a rare (yet required) character because it's not likely to be found in the string, and a quick check of the `study` list that turns up nothing means a match can be discounted immediately without having to rescan the entire string.

If the rare character *is* found in the string, and if that character must occur at a known position in any possible match (such as with ⌈`..this`⌋, but not ⌈`.?this`⌋), the transmission can use the `study` list to start matching from near the location. This saves time by bypassing perhaps large portions of the string.*

When not to use study

- Don't use `study` when the target string is short. In such cases, the normal fixed-string cognizance optimization should suffice.

- Don't use `study` when you plan only a few matches against the target string (or, at least, few before it is modified, or before you `study` a different string). An overall speedup is more likely if the time spent to `study` a string is amortized over many matches. With just a few matches, the time spent scanning the string (to build the `study` list) can overshadow any savings.

  Note that with the current implementation, `m/…/g` is considered one match: the `study` list is consulted only with the first attempt. Actually, in the case of a scalar context `m/…/g`, it *is* consulted with each match, but it reports the same location each time for all but the first match, that location is before the beginning of the match, so the check is just a waste of time.

- Don't use `study` when Perl has no *literal text cognizance* (☞ 286) for the regular expressions that you intend to benefit from the `study`. Without a known character that must appear in any match, `study` is useless.

---

* There's a bug in the current implementation which disables this optimization when the regex begins with literal text. This is unfortunate because such expressions have generally been thought to benefit most from `study`.

When study can help

`study` is best used when you have a large string you intend to match many times before the string is modified. A good example is a filter I used in preparing this book. I write in a home-grown markup that the filter converts to SGML (which is then converted to *troff* which is then converted to PostScript). Within the filter, an entire chapter eventually ends up within one huge string (this chapter is about 650 kilobytes). Before exiting, I apply a bevy of checks to guard against mistaken markup leaking through. These checks don't modify the string, and they often look for fixed strings, so they're what `study` thrives on.

Study in the real world

It seems that `study` has been hexed from the start. First, the programming populace never seemed to understand it well. Then, a bug in Perl versions 5.000 and 5.001 rendered `study` completely useless. In recent versions, that's been fixed, but now there's a `study` bug that can cause successful matches in `$_` to fail (even matches that have nothing to do with the string that was `study`ed). I discovered this bug while investigating why my markup filter wasn't working, quite coincidentally, just as I was writing this section on `study`. It was a bit eerie, to say the least.

You can get around this bug with an *explicit* `undef`, or other modification, of the `study`ed string (when you're done with it, of course). The automatic assignment to `$_` in `while (<>)` is *not* sufficient.

When `study` can work, it often doesn't live up to its full potential, either due to simple bugs or an implementation that hasn't matured as fast as the rest of Perl. At this juncture, I recommend against the use of `study` unless you have a very specific situation you know benefits. If you do use it, and the target string is in `$_`, be sure to `undef`ine it when you are done.

Putting It All Together

This chapter has gone into great detail about Perl's regular expression flavor and operators. At this point, you may be asking yourself what it all means it may take a fair amount of use to "internalize" the information.

Let's look again at the initial CSV problem. Here's my Perl5 solution, which, as you'll note, is fairly different from the original on page 205:

```
    @fields = ();
    push(@fields, $+) while $text =~ m{
        "([^"\\]*(?:\\.[^"\\]*)*)",?      # standard quoted string, with possible comma
       | ([^,]+),?                        # anything else, with possible comma
       | ,                                # lone comma
    }gx;

    # add afinal empty field if there's a trailing comma
    push(@fields, undef) if substr($text,-1,1) eq ',';
```

Like the first version, it uses a scalar-context `m/…/g` with a while loop to iterate over the string. We want to stay in synch, so we make sure that at least one of the alternatives matches at any location a match could be started. We allow three types of fields, which is reflected in the three alternatives of the main match.

Because Perl5 allows you to choose exactly which parentheses are capturing and which aren't, we can ensure that after any match, `$+` holds the desired text of the field. For empty fields where the third alternative matches and no capturing parentheses are used, `$+` is guaranteed to be `undef`ined, which is exactly what we want. (Remember `undef` is different from an empty string returning these different values for empty and `""` fields retains the most information.)

The final `push` covers cases in which the string ends with a comma, signifying a trailing empty field. You'll note that I don't use `m/,$/` as I did earlier. I did so earlier because I was using it as an example to show regular expressions, but there's really no need to use a regex when a simpler, faster method exists.

Along with the CSV question, many other common tasks come up time and again in the Perl newsgroups, so I'd like to finish out this chapter by looking at a few of them.

Stripping Leading and Trailing Whitespace

By far the best all-around solution is the simple and obvious:

```
    s/^\s+//;
    s/\s+$//;
```

For some reason, it seems to be The Thing to try to find a way to do it all in one shot, so I'll offer a few methods. I don't recommend them, but it's educational to understand why they work, and why they're not desirable.

`s/\s*(.*?)\s*$/$1/`

> Commonly given as a great example of the non-greediness that is new in Perl5, but it's not a great example because it's so much slower (by about three times in my tests) than most of the other solutions. The reason is that with each character, before allowing dot to match, the `*?` must try to see whether what follows can match. That's a lot of backtracking, particularly since it's the kind that goes in and out of the parentheses (☞ 151).

`s/^\s*(.*\S)?\s*$/$1/`

> Much more straightforward. The leading ⌈`^\s*`⌋ takes care of leading whitespace before the parentheses start capturing. Then the ⌈`.*`⌋ matches to the end of the line, with the `\s` causing backtracking past trailing whitespace to the final non-whitespace. If there's nothing but whitespace in the first place, the ⌈`(.*\S)?`⌋ fails (which is fine), and the final ⌈`\s*`⌋ zips to the end.

`$_ = $1 if m/^\s*(.*\S)?/`

> More or less the same as the previous method, this time using a match and assignment instead of a substitution. With my tests, it's about 10 percent faster.

`s/^\s*|\s*$//g`

> A commonly thought-up solution that, while not incorrect, has top-level alternation that removes many of the optimizations that might otherwise be possible. The `/g` modifier allows each alternative to match, but it seems a waste to use `/g` when we know we intend at most two matches, and each with a different subexpression. Fairly slow.

Their speed often depends on the data being checked. For example, in rare cases when the strings are very, very long, with relatively little whitespace at either end, **`s/^\s+//; s/\s+$//`** can take twice the time of **`$_ = $1 if m/^\s*(.*\S)?/`**. Still, in my programs, I use **`s/^\s+//;  s/\s+$//`** because it's almost always fastest, and certainly the easiest to understand.

Adding Commas to a Number

People often ask how to print numbers with commas, as in `12,345,678`. The FAQ currently gives

```
1 while s/^(-?\d+)(\d{3})/$1,$2/;
```

which repeatedly scans to the last (non-comma'd) digit with ⌈`\d+`⌋, backtracks three digits so ⌈`\d{3}`⌋, can match, and finally inserts a comma via the replacement text '`$1`, `$2`'. Because it works primarily "from the right" instead of the normal left, it is useless to apply with `/g`. Thus, multiple passes to add multiple commas is achieved using a `while` loop.

You can enhance this solution by using a common optimization from Chapter 5 (☞ 156), replacing ⌈`\d{3}`⌋ with ⌈`\d\d\d`⌋. Why bother making the regex engine count the occurrences when you can just as easily say exactly what you want? This one change saved a *whopping* three percent in my tests. (A penny saved. . .)

Another enhancement is to remove the start-of-string anchor. This allows you to comma-ify a number (or numbers) within a larger string. As a byproduct, you can then safely remove the ⌈`-?`⌋, since it exists only to tie the first digit to the anchor. This change could be dangerous if you don't know the target data, since `3.14159265` becomes `3.14,159,265`. In any case, if you know the number is the string by itself, the anchored version is better.

A completely different, but almost-the-same approach I've come up with uses a single `/g`-governed substitution:

```
s<
    (\d{1,3})           #  before a comma: one to three digits
    (?=                 #  followed by, but not part of what's matched . . .
       (?:\d\d\d)+      #          some number of triplets . . .
       (?!\d)           #          . . . not followed by another digit
    )                   #          (in other words, which ends the number)
>< $1, >gx;
```

Because of the comments and formatting, it might look more complex than the FAQ solution, but in reality it's not so bad, and is a full third faster. However, because it's not anchored to the start of the string, it faces the same problem with `3.14159265`. To take care of that, and to bring it in line with the FAQ solution for all strings, change the ⌈`(\d{1,3})`⌋ to ⌈`\G((?:^-)?\d{1,3})`⌋. The `\G` anchors the overall match to the start of the string, and anchors each subsequent `/g`-induced match to the previous one. The ⌈`(?:^-)?`⌋ allows a leading minus sign at the start of the string, just as the FAQ solution does. With these changes, it slows down a tad, but my tests show it's still over 30 percent faster than the FAQ solution.

Removing C Comments

It's challenging to see how crisply you can strip C comments from text. In Chapter 5, we spent a fair amount of time coming up with the general comment-matching ⌈`/\*[^*]*\*+([^/*][^*]*\*+)*/`⌋, and a Tcl program to remove comments. Let's express it in Perl.

Chapter 5 dealt with generic NFA engines, so our comment-matching regex works fine in Perl. For extra efficiency, I'd use non-capturing parentheses, but that's about the only direct change I'd make. It's not unreasonable to use the FAQ's simpler ⌈`/\*.*?\*/`⌋ Chapter 5's solution leads the engine to a match more efficiently, but ⌈`/\*.*?\*/`⌋ is fine for applications that aren't time critical. It's certainly easier to understand at first glance, so I'll use it to simplify the first draft of our comment-stripping regex.

Here it is:

```
    s{
       #  First, we'll list things we want to match, but not throw away
       (
           " (\\.|[^"\\])* "   #  doublequoted string.
         |                     #  -or-
           ' (\\.|[^'\\])* '   #  singlequoted constant
       )
    |  #  OR.. . .
       #  . . . we'll match a comment. Since it's not in the $1 parentheses above,
       #  the comments will disappear when we use $1 as the replacement text.
       /\*  .*?  \*/           #  Traditional C comments.
       |                       #  -or-
       //[^\n] *               #  C++  //-style comments
    }{$1}gsx;
```

After applying the changes discussed during the Tcl treatment, and combining the two comment regexes into one top-level alternative (which is easy since we're writing the regex directly and not building up from separate `$COMMENT` and `$COMMENT1` components), our Perl version becomes:

```
    s{
        #  First, we'll list things we want to match, but not throw away
        )
           [^"'/]+                                          #  other stuff
         |                                                  #  -or-
           (?:"[^"\\]*(?:\\.[^"\\]*)*"  [^"'/]*)+    #
doublequotedstring.
         |                                                  #  -or-
           (?:'[^'\\]*(?:\\.[^'\\]*)*'  [^"'/]*)+    #
singlequotedconstant
        )
    |  #  OR . . .
        #  . . . we'll match a comment. Since it's not in the $1 parentheses above,
        #  the comments will disappear when we use $1 as the replacement text.

        / (?:                                   #  (all comments start with a
slash)
            \*[^*]*\*+(?:[^/*][^*]*\*+)*/   #  Traditional C comments.
            |                                   #  -or-
            /[^\n]*                             #  C++  //-style comments
          )
    }{$1}gsx;
```

With the same tests as Chapter 5's Tcl version, times hover around 1.45 seconds (compare to Tcl's 2.3 seconds, the first Perl's version at around 12 seconds, and Tcl's first version at around 36 seconds).

To make a full program out of this, just insert it into:

```
undef $/;                      # Slurp-whole-file mode
$_ = join('', <>);             # The join (…) can handle multiple files.
... insert the substitute command from above ...
print;
```

Yup, that's the whole program.

Matching an Email Address

I'd like to finish with a lengthy example that brings to bear many of the regex techniques seen in these last few chapters, as well as some extremely valuable lessons about building up a complex regular expression using variables. Verifying correct syntax of an Internet email address is a common need, but unfortunately, because of the standard's complexity,* it is quite difficult to do simply. In fact, it is impossible with a regular expression because address comments may be nested. (Yes, email addresses can have comments: comments are anything between parentheses.) If you're willing to compromise, such as allowing only one level of nesting in comments (suitable for any address I've ever seen), you can take a stab at it. Let's try.

Still, it's not for the faint at heart. In fact, the regex we'll come up with is *4,724* bytes long! At first thought, you might think something as simple as ⌜`\w+\@[.\w]+`⌟ could work, but it is *much* more complex. Something like

```
    Jeffy <"That Tall Guy"@ora.com (this address no longer
active)>
```

is perfectly valid as far as the specification is concerned.** So, what constitutes a lexically valid address? Table 7-11 lists a lexical specification for an Internet email address in a hybrid BNF/regex notation that should be mostly self-explanatory. In addition, comments (item 22) and whitespace (spaces and tabs) are allowed between most items. Our task, which we choose to accept, is to convert it to a regex as best we can. It will require every ounce of technique we can muster, but it is possible.***

Levels of interpretation

When building a regex using variables, you must take extra care to understand the quoting, interpolating, and escaping that goes on. With ⌜`^\w+\@[.\w]+$`⌟ as an example, you might naively render that as

```
    $username = "\w+";
    $hostname = "\w+(\.\w+)+";
    $email    = "^$username\@$hostname$";
    ⋮
    ... m/$email/o ...
```

but it's not so easy. While evaluating the doublequoted strings in assigning to the variables, the backslashes are interpolated and discarded: the final `$email` sees '`^w+@w+(.w+)+$`' with Perl4, and can't even compile with Perl5 because of the trailing dollar sign. Either the escapes need to be escaped so they'll be preserved

---

[*] Internet RFC 822. Available at:
`ftp://ftp.rfc-editor.org/in-notes/rfc822.txt`

[**] It is certainly not valid in the sense that mail sent there will bounce for want of an active username, but that's an entirely different issue.

[***] The program we develop in this section is available on my home page—see Appendix A.

Table 7-11: Somewhat Formal Description of an Internet Email Address

	Item	Description
1	mailbox	addr-spec | phrase route-addr
2	addr-spec	local-part @ domain
3	phrase	(word)+
4	route-addr	<(route)? addr-spec >
5	local-part	word(.word)*
6	domain	sub-domain(.sub-domain)*
7	word	atom | quoted-string
8	route	@ domain (,@domain)*:
9	sub-domain	domain-ref | domain-literal
10	atom	(any char except specials, space and ctls)+
11	quoted-string	"(qtext | quoted-pair)*"
12	domain-ref	atom
13	domain-literal	[(dtext | quoted-pair)*]
14	char	any ASCII character (000-177 octal)
15	ctl	any ASCII control (000-037 octal)
16	space	ASCII space (040 octal)
17	CR	ASCII carriage return (015 octal)
18	specials	any of the characters: ()<>@,;:\".[]
19	qtext	any char except ", \ and CR
20	dtext	any char except [,], \ and CR
21	quoted-pair	\ char
22	comment	((ctext | quoted-pair | comment)*)
23	ctext	any char except (,), \ and CR

through to the regex, or a singlequoted string must be used. Singlequoted strings are not applicable in all situations, such as in the third line where we really do need the variable interpolation provided by a doublequoted string:

```
$username = '\w+';
$hostname = '\w+(\.\w+)+';
$email    = "^$username\@$hostname\$";
```

Let's start building the real regex by looking at item 16 in Table 7-11. The simple **`$space = "•"`** isn't good because if we use the `/x` modifier when we apply the regex (something we plan to do), spaces outside of character classes, such as this one, will disappear. We can also represent a space in the regex with ⌜`\040`⌟ (octal `40` is the ASCII code for the space character), so we might be tempted to assign `"\040"` to `$space`. This would be a silent mistake because, when the doublequoted string is evaluated, `\040` is turned into a space. This is what the regex will see, so we're right back where we started. We want the regex to see `\040` and turn it into a space itself, so again, we must use `"\\040"` or '`\040'`.

Getting a match for a literal backslash into the regex is particularly hairy because it's also the regex escape metacharacter. The regex requires ⌜`\\`⌟ to match a single literal backslash. To assign it to, say, `$esc`, we'd like to use `'\\'`, but because `\\`

is special even within singlequoted strings,* we need **`$esc = '\\\\'`** just to have the final regex match a single backslash. This backslashitis is why I make `$esc` once and then use it wherever I need a literal backslash in the regex. We'll use it a few times as we construct our address regex. Here are the preparatory variables I'll use this way:

```
    #  Some things for avoiding backslashitis later on.
    $esc         =  \\\\';             $Period      =
    $space       = '\040';             $tab         = '\t';
    $OpenBR      = '\[;                $CloseBR     =
    $OpenParen   = '\($;               $CloseParen  = '\)';
    $NonASCII    = '\x80-\xff';        $ctrl        = '\000-\037';
    $CRlist      = '\n\015';   #  note: this should really be only\ 015.
```

The `$CRlist` requires special mention. The specification indicates only the ASCII carriage return (octal `015`). From a practical point of view, this regex is likely to be applied to text that has already been converted to the system-native newline format where `\n` represents the carriage return. This may or may not be the same as an ASCII carriage return. (It usually is, for example, on MacOS, but not with Unix; ☞ 72.) So I (perhaps arbitrarily) decided to consider both.

Filling in the basic types

Working mostly from Table 7-11, bottom up, here are a few character classes we'll be using, representing items 19, 20, 23, and a start on 10:

```
    #  Items 19, 20, 21
    $qtext = qq/[^$esc$NonASCII$CRlist"]/;                 #  for
within  "..."
    $dtext = qq/[^$esc$NonASCII$CRlist$OpenBR$CloseBR]/; #  for
within  [...]
    $quoted_pair = qq< $esc [^$NonASCII] >; #  an escaped character

    #  Item 10: atom
    $atom_char =
qq/[^($space)<>\@,;:".$esc$OpenBR$CloseBR$ctrl$NonASCII]/;
    $atom = qq<
      $atom_char+      #  some number of atom characters. . .
      (?!$atom_char) #  . .not followed by something that could be part of an atom
    >;
```

That last item, `$atom`, might need some explanation. By itself, `$atom` need be only `$atom_char+`, but look ahead to Table 7-11's item 3, `phrase`. The combination yields ⌈`($atom_char+)+`⌋, a lovely example of one of those neverending-match patterns (☞ 144). Building a regex in a variable is prone to this kind of hidden danger because you can't normally see everything at once. This visualization problem is why I used **`$NonASCII = '\x80-\xff'`** above. I could have used **`"\x80-\xff"`**, but I wanted to be able to print the partial regex at any time during testing. In the latter case, the regex holds the raw bytes—fine for the regex engine, but not for our display if we print the regex while debugging.

---

[*] Within Perl singlequoted strings, `\\` and the escaped closing delimiter (usually `\'`) are special. Other escapes are passed through untouched, which is why `\040` results in `\040`.

Getting back to ⌈`($atom_char+)+`⌋, to help delimit the inner-loop single atom, I can't use ⌈`\b`⌋ because Perl's idea of a word is completely different from an email address atom. For example, '`--genki--`' is a valid `atom` that doesn't match ⌈`\b$atom_char+\b`⌋. Thus, to ensure that backtracking doesn't try to claim an atom that ends in the middle of what it should match, I use `(?!…)` to make sure that `$atom_char` can't match just after the `atom`'s end. (This is a situation where I'd really like the *possessive quantifiers* that I pined for in the footnote on page 111.)

Even though these are doublequoted strings and not regular expressions, I use free spacing and comments (except within the character classes) because these strings will eventually be used with an `/x`-governed regex. But I do take particular care to ensure that each comment ends with a newline, as I don't want to run into the overzealous comment problem (☞ 223).

Address comments

Comments with this specification are difficult to match because they allow nested parentheses, something impossible to do with a single regular expression. You can write a regular expression to handle up to a certain number of nested constructs, but not to an arbitrary level. For this example, I've chosen to implement `$comment` to allow for one level of internal nesting:

```
    #  Items 22 and 23, comment.
    #  Impossible to do properly with a regex, I make do by allowing at most one level of
nesting.
    $ctext   = qq< [^$esc$NonASCII$CRlist()] >;
    $Cnested = qq< $OpenParen (?: $ctext | $quoted_pair )*
$CloseParen >;
    $comment = qq< $OpenParen
                         (?: $ctext | $quoted_pair | $Cnested )*
                   $CloseParen >;
    $sep     = qq< (?: [$space$tab] | $comment )+ >; #  required
separator
    $X       = qq< (?: [$space$tab] | $comment )* >; #
optionalseparator
```

You'll not find `comment`, item 22, elsewhere in the table. What the table doesn't show is that the specification allows comments, spaces, and tabs to appear freely between most tokens. Thus, we create `$x` for optional spaces and comments, `$sep` for required ones.

The straightforward bulk of the task

Most items in Table 7-11 are relatively straightforward to implement. One trick is to make sure to use `$x` where required, but for efficiency's sake, no more often than necessary. The method I use is to provide `$x` only between elements within a single subexpression. Most of the remaining items are shown on the next page.

```
    #  Item 11: doublequoted string, with escaped items allowed
    $quoted_str = qq<
            " (?:                         #  opening quote . . .
                  $qtext                  #     Anything except backslash
and quote
                  |                       #      or
                  $quoted_pair            #     Escaped something
(something != CR)
                                 )* "     #  closing quote
    >;

    #  Item 7: word is an atom or quoted string
    $word = qq< (?: $atom | $quoted_str ) >;

    #  Item 12: domain-ref is just an atom
    $domain_ref = $atom;

    #  Item 13 domain-literal is like a quoted string, but [. . .] instead of ". . ."
    $domain_lit = qq<  $OpenBR                              # [
                       (?: $dtext | $quoted_pair )*    #      stuff
                       $CloseBR
#             ]
    >;

    #  Item 9: sub-domain is a domain-refor domain-literal
    $sub_domain = qq< (?: $domain_ref | $domain_lit ) >;


    #  Item 6: domain is a list of subdomains separated by dots.
    $domain = qq< $sub_domain                       #  initial subdomain
                   (?:                              #
                      $X $Period                    #  if led by aperiod. . .
                      $X $sub_domain                #    . . .further okay
                   )*
    >;

    #  Item 8: a route. A bunch of "@ $domain" separated by commas, followed by a colon
    $route = qq< \@ $X $domain
                 (?: $X , $X \@ $X $domain )*  #  further okay, if led by
comma
                                              #  closing colon
    >;

    #  Item 5: local-part is a bunch of word separated by periods
    $local_part = qq< $word                        #  initial word
            (?: $X $Period $X $word )*         #  further okay, if led by
```

```
aperiod
    >;

    #  Item 2: addr-spec is local@domain
    $addr_spec  = qq< $local_part $X \@ $X $domain >;

    #  Item 4: route-addr is <route? addr-spec>
    $route_addr = qq[ < $X                        #  leading <
                      (?: $route $X )?            #        optional route
                          $addr_spec              #        address spec
                                      $X >        #
trailing >
    ];
```

Item 3 —phrase

phrase poses some difficulty. According to Table 7-11, it is one or more `word`, but we can't use **`(?:$word)+`** because we need to allow `$sep` between items. We can't use **`(?:$word|$sep)+`**, as that doesn't *require* a `$word`, but merely *allows* one. So, we might be tempted to try **`$word(?:$word|$sep)*`**, and this is where we really need to keep our wits about us. Recall how we constructed `$sep`. The non-comment part is effectively `[$space$tab]+`, and wrapping this in the new `(...)*` smacks of a neverending match (☞ 166). The `$atom` within `$word` would also be suspect except for the `(?!...)` we took care to tack on to checkpoint the match. We could try the same with `$sep`, but I've a better idea.

Four things are allowed in a `phrase`: quoted strings, atoms, spaces, and comments. Atoms are just sequences of `$atom_char` f these sequences are broken by spaces, it means only that there are multiple atoms in the sequence. We don't need to identify individual atoms, but only the extent of the entire sequence, so we can just use something like:

```
    $word (?: [$atom_char$space$tab] | $quoted_string | $comment
)+
```

We can't actually use that character class because `$atom_char` is already a class itself, so we need to construct a new one from scratch, mimicking `$atom_char`, but removing the space and tab (*removing* from the list of a negated class *includes* them in what the class can match):

```
    # Item 3: phrase
    $phrase_ctrl = '\000-\010\012-\037';  # like ctrl, but
without tab

    # Like atom-char, but without listing space, and uses phrase_ctrl.
    # Since the class is negated, this matches the same as atom-charplus space and tab
    $phrase_char =

qq/[^()<>\@,;:".$esc$OpenBR$CloseBR$NonASCII$phrase_ctrl]/;
    $phrase = qq< $word                # one word, optionally followed by .
..
                  (?:
                      $phrase_char     |  # atom and spaceparts, or . . .
                      $comment         |  # comments, or . . .
                      $quoted_str         # quoted strings
                   )*
    >;
```

Unlike all the other constructs so far, this one matches trailing whitespace and comments. That's not bad, but for efficiency's sake, we remember we don't need to insert `$x` after any use of `$phrase`.

Wrapping up with `mailbox`

Finally, we need to address item 1, the simple:

```
#  Item #1: mailbox is an addr_spec or a phrase/route_addr
$mailbox = qq< $X                       #  optional leading comment
                (?: $addr_spec          #  address
                    |                   #    or
                    $phrase $route_addr #  name and address
                ) $X                    #  optional trailing comment
>;
```

Whew, done*!*

Well, we can now use this like:

```
    die "invalid address [$addr]\n" if $addr !~ m/^$mailbox$/xo;
```

(With a regex like this, I *strongly* suggest not forgetting the `/o` modifier.)*

It might be interesting to look at the final regex, the contents of `$mailbox`. After removing comments and spaces and breaking it into lines for printing, here are the first few out of 60 or so lines:

```
(?:[\040\t]|\((?:[^\\\x80-\xff\n\015()]|\\[^\x80-\xff]|\((?:[^\\\x80-\xff\n\015(

)]|\\[^\x80-\xff])*\))*\))*(?:(?:[^(\040)<>@,;:".\\\[\]\000-\037\x80-\xff]+(?![^

(\040)<>@,;:".\\\[\]\000-\037\x80-\xff])|"(?:[^\\\x80-\xff\n\015"]|\\[^\x80-\xff

])*")(?:(?:[\040\t]|\((?:[^\\\x80-\xff\n\015()]|\\[^\x80-\xff]|\((?:[^\\\x80-\xf

f\n\015()]|\\[^\x80-\xff])*\))*\))*\.(?:[\040\t]|\((?:[^\\\x80-\xff\n\015()]|\\[

^\x80-\xff]|\((?:[^\\\x80-\xff\n\015()]|\\[^\x80-\xff])*\))*\))*(?:[^(\040)<>@,;

:".\\\[\]\000-\037\x80-\xff]+(?![^(\040)<>@,;:".\\\[\]\000-\037\x80-\xff])|"(?:
[
```

Wow. At first you might think that such a gigantic regex could not possibly be efficient, but the size of a regex has little to do with its efficiency. More at stake is how much backtracking it has to do. Are there places with lots of alternation? Neverending-match patterns and the like? Like an efficient set of ushers at the local 20-screen theater complex, a huge regex can still guide the engine to a fast match, or to a fast failure, as the case may be.

Recognizing shortcoming

To actually use a regex like this, you need to know its limitations. For example, it recognizes only Internet email addresses, not local addresses. While logged in to my machine, `jfriedl` by itself is a perfectly valid email address, but is not an Internet email address. (This is not a problem with the regex, but with its use.) Also, an address might be lexically valid but might not actually point anywhere, as with the earlier `That Tall Guy` example. A step toward eliminating some of these is to require a domain to end in a two- or three-character subdomain (such as `.com or .jp`). This could be as simple as appending **`$esc . $atom_char {2,3}`** to `$domain`, or more strictly with something like:

```
    $esc . (?: com | edu | gov | … | ca | de | jp | u[sk] … )
```

When it comes down to it, there is *absolutely no way* to ensure a particular address actually reaches someone. Period. Sending a test message is a good indicator if someone happens to reply. Including a `Return-Receipt-To` header in the message is also useful, as it has the remote system generate a short response to the effect that your message has arrived to the target mailbox.

---

* From the "don't try this at home, kids" department: During initial testing, I was stumped to find that the optimized version (presented momentarily) was consistently slower than the normal version. I was really dumbfounded until I realized that I'd forgotten `/o`! This caused the entire huge regex operand to be reprocessed for each match **(**☞ 268). The optimized expression turned out to be considerably longer, so the extra processing time completely overshadowed any regex efficiency benefits. Using `/o` not only revealed that the optimized version was faster, but caused the whole test: to finish an order of magnitude quicker.

Optimizations—unrolling loops

As we built our regex, I hope that you recognized ample room for optimizations. Remembering the lessons from Chapter 5, our friend the quoted string is easily unrolled to

```
    $quoted_str = qq< "                                  # opening
quote
                      $qtext *                           #   leading
normal
                      (?: $quoted_pair $qtext * )*       #
(special normal*)*
                    "                                    # closing
quote
    >;
```

while `$phrase` might become:

```
    $phrase = qq< $word                            # leading word
                  $phrase_char *                   # "normal"atoms
and/orspaces
               (?:
                  (?: $comment | $quoted_str )     # "special"comment or
quoted string
                  $phrase_char *                   # more "normal"
               )*
    >
```

Items such as `$Cnested`, `$comment`, `$phrase`, `$domain_lit`, and `$x` can be optimized similarly, but be careful—some can be tricky. For example, consider `$sep` from the section on comments. It requires at least one match, but using the normal unrolling-the-loop technique creates a regex that doesn't require a match.

Talking in terms of the general unrolling-the-loop pattern (☞ 164), if you wish to require *special*, you can change the outer `(…)*` to `(…)+`, but that's not what `$sep` needs. It needs to require something, but that something can be either *special* or *normal*.

It's easy to create an unrolled expression that requires one or the other in particular, but to require either we need to take a dual-pronged approach:

```
$sep = qq< (?:
              [$space$tab +                                  #  for when
space is first
              (?: $comment [$space$tab]* )*
            |
              (?: $comment [$space$tab]* )+                  #  for when
comment is first
            )
>;
```

+This contains two modified versions of the ⌜`normal* (special normal*)*`⌟ pattern, where the class to match spaces is *normal*, and `$comment` is *special*. The first requires spaces, then allows comments and spaces. The second requires a comment, then allows spaces. For this last alternative, you might be tempted to consider `$comment` to be *normal* and come up with:

```
$comment  (?: [$space$tab]+ $comment )*
```

This might look reasonable at first, but that plus is the quintessence of a neverending match. Removing the plus fixes this, but the resulting regex loops on each space—not a pillar of efficiency.

As it turns out, though, none of this is needed, since `$sep` isn't used in the final regex; it appeared only in the early attempt of   . I kept it alive this long because this is a common variation on the unrolling-the-loop pattern, and the discussion of its touchy optimization needs is valuable.

Optimizations—flowing spaces

Another kind of optimization centers on the use of `$X`. Examine how the `$route` part of our regex matches '`@•gateway•:`'. You'll find times where an optional part fails, but only after one or more internal `$x` match. Recall our definitions for `$domain` and `$route`:

```
# Item 6: domain is a list ofsubdomains separated by dots.
$domain = qq< $sub_domain              # initial subdomain
              (?:                      #
                 $X $Period            # if led by a period . . .
                 $X $sub_domain        #    . . . further okay
               )*
>;

 # Item 8: a route. A bunch of "@ $domain" separated by commas, followed by a colon
 $route = qq< \@ $X $domain
              (?: $X , $X \@ $X $domain )*  # further okay, if led by
comma
          :                                 # closing colon
 >;
```

After the `$route` matches the initial '`@•gateway•:`' and the first `$sub_domain` of `$domain` matches '`@•gateway•:`', the regex checks for a period and another `$sub_domain` (allowing `$X` at each juncture). In making the first attempt at this **`$X $Period $X $sub_domain`** subexpression, the initial `$X` matches the space '`@•gateway•:`', but the subexpression fails trying to match a period. This causes backtracking out of the enclosing parentheses, and the instance of `$domain` finishes.

Back in `$route`, after the first `$domain` is finished, it then tries to match another if separated by a colon. Inside the **`$X , $X \@`**… subexpression, the initial `$X` matches the same space that had been matched (and unmatched) earlier. It, too, fails just after.

It seems wasteful to spend the time matching `$X` when the subexpression ends up failing. Since `$X` can match almost anywhere, it's more efficient to have it match only when we know the associated subexpression can no longer fail.

Consider the following, whose only changes are the placement of `$X`:

```
$domain = qq<
     $sub domain $X
     (?:
        $Period $X $sub_domain $X
     )*
>;
```

```
$route = qq<
    \@ $X $domain
    (?: , $X \@ $X $domain)*
    : $X
>;
```

Here, we've changed the guideline from "use `$X` only between elements within a subexpression" to "ensure a subexpression consumes any trailing `$X`." This kind of change has a ripple effect on where `$X` appears in many of the expressions.

After applying all these changes, the resulting expression is almost 50 percent longer (these lengths are after comments and free spacing are removed), but executed 9-19 percent faster with my benchmarks (the 9 percent being for tests that primarily failed, 19 percent for tests that primarily matched). Again, using `/o` is very important. The final version of this regex is in Appendix B.

Building an expression through variables—summary

This has been a long example, but it illustrates some important points. Building a complex regex using variables is a valuable technique, but must be used with skill and caution. Some points to keep in mind include:

- Pay heed to what is interpolated by whom, and when, with specific attention given to the many personalities of the backslash escape.

- Be careful about using alternation in a subexpression without enclosing parentheses. A mistake like

```
$word       = qq< $atom | $quoted_str >;
$local_part = qq< $word (?: $X $Period $X $word) * >
```

is easy to make, but difficult to recognize. So, use parentheses liberally, and for speed, use `(?:…)` when possible. If you don't easily recognize the mistake in the snippet above, consider what `$line` becomes in:

```
$field = "Subject|From|Date";
$line  = "^$field: (.*)";
```

⌜`^Subject|From|Date:•(.*)`⌟ is very different from, and certainly not as useful as, ⌜`^(Subject|From|Date):•(.*)`⌟.

- Remember that `/x` does not affect character classes, so free spacing and comments can't be used within them.

- Regex `#`-style comments continue until newline or the end of the regex. Consider:

```
$domain_ref = qq< $atom  # just a simple atom >
$sub_domain = qq< (?: $domain_ref | $domain_lit ) >
```

Just because the *variable* `$domain_ref` ends, it doesn't mean that the comment that's inserted into the regex *from it* ends. The comment continues until the end of the *regex* or until a newline, so here the comment extends past the end of `$domain_ref` to consume the alternation and the rest of `$sub_domain`, and anything that follows `$sub_domain` wherever it is used, until the end of the regex or a newline. This can be averted by adding the newline manually (☞ 223): `$domain_ref = qq< $atom # just a simple atom\n >`

- Because it is often useful to print a regex during debugging, consider using something like `'\0xff'` instead of `"\0xff"`.

- Understand the differences among:

```
$quoted_pair = qq< $esc[^$NonASCII] >;
$quoted_pair = qq< $esc_[^$NonASCII] >;
$quoted_pair = qq< ${esc}[^$NonASCII] >;
```

The first is *very* different from the other two. It is interpreted as an attempt to index an element into the array `@esc`, which is certainly not what is wanted here (☞ 222).

Final Comments

I'm sure it's obvious that I'm quite enamored with Perl's regular expressions, and as I noted at the start of the chapter, it's with good reason. Larry Wall, Perl's creator, apparently let himself be ruled by common sense and the Mother of Invention. Yes, the implementation has its warts, but I still allow myself to enjoy the delicious richness of Perl's regex language.

However, I'm not a blind fanatic—Perl does not offer features that I wish it did. The most glaring omission is offered by other implementations, such as by Tcl, Python, and GNU Emacs: the index into the string where the match (and `$1`, `$2`, etc.) begins and ends. You can get a copy of text matched by a set of parentheses using the aforementioned variables, but in general, it's impossible to know exactly where in the string that text was taken from. A simple example that shows the painfulness of this feature's omission is in writing a regex tutor. You'd like to show the original string and say "The first set of parentheses matched right here, the second set matched here, and so on," but this is currently impossible with Perl.

Another feature I've found an occasional need for is an array **`($1, $2, $3, …)`** similar to Emacs' `match-data` (☞ 196). I can construct something similar myself using:

```
$parens[0] = $&;
$parens[1] = $1;
$parens[2] = $2;
$parens[3] = $3;
$parens[4] = $4;
   ⋮
```

but it would be nicer if this functionality were built-in.

Then there are those *possessive quantifiers* that I mentioned in the footnote on page 111. They could make many expressions much more efficient.

There are a lot of esoteric features I can (and do) dream of. One that I once went so far as to implement locally was a special notation whereby the regex would reference an associative array *during the match,* using ⌈`\1`⌋ and such as an index. It made it possible to extend something like ⌈`(['"]).*?\1`⌋ to include <…>, … and the like.

Another feature I'd love to see is named subexpressions, similar to Python's *symbolic group names* feature. These would be capturing parentheses that (somehow) associated a variable with them, filling the variable upon a successful match. You could then pick apart a phone number like (inventing some fictitious ⌈`(?<var>…)`⌋ notation on the fly):

⌈`(?<$area>\d\d\d)-(?<$exchange>\d\d\d)-(?<$num>\d\d\d\d)`⌋

Well, I'd better stop before I get carried away. The sum of it all is that I definitely do not think Perl is the ideal regex-wielding language.

But it is very close.

<><><><><><><><><><>

Notes for Perl4

For the most part, regex use flows seamlessly from Perl4 to Perl5. Perhaps the largest backward-compatibility issue is that @ now interpolates within a regex (and a doublequoted string, for that matter). Still, you should be aware of a number of subtle (and not-so-subtle) differences when working with Perl4:

Perl4 Note #1

Page 217 The special variables `$&`, `$1`, and so on are not read-only in Perl4 as they are in Perl5. Although it would be useful, modifying them does not magically modify the original string they were copied from. For the most part, they're just normal variables that are dynamically scoped and that get new values with each successful match.

Perl4 Note #2

Page 217 Actually, with Perl4, `` $` `` sometimes does refer to the text from the start of the match (as opposed to the start of the string). A bug that's been fixed in newer versions caused `` $` `` to be reset each time the regex was compiled. If the regex operand involved variable interpolation, and was part of a scalar-context `m/…/g` such as the iterator of a `while` loop, this recompilation (which causes `` $` `` to be reset) is done during each iteration.

Perl4 Note #3

Page 218 In Perl4, `$+` magically becomes a copy of `$&` when there are no parentheses in the regex.

Perl4 Note #4

Page 220 Perl4 interpolates `$MonthName[…]` as an array reference only if `@MonthName` is known to exist. Perl5 does it regardless.

Perl4 Note #5

Page 222 The escape of an escaped closing delimiter is *not* removed in Perl4 as it is in Perl5. It matters when the delimiter is a metacharacter. The (rather farfetched) substitution `s*2\*2*4*` would not work as expected in Perl5.

Perl4 Note #6

Page 247 Perl4 allows you to use whitespace as a match-operand delimiter. Although using newline, for example, was sometimes convenient, for the most part I'd leave this maintenance nightmare for an Obfuscated Perl contest.

Perl4 Note #7

Page 247 Remember, in Perl4, *all* escapes are passed through. (See note #5.)

Perl4 Note #8

Page 247 Perl4 does not support the four `m{…}` special-case delimiters for the match operator. It does, however, support them for the substitution operator.

Perl4 Note #9

Page 247 Perl4 supports the special `?`-match, but only in the `?…?` form. The `m?…?` form is not special.

Perl4 Note #10

Page 247 With Perl4, a call to any `reset` in the program resets all `?`-delimited matches. Perl5's `reset` affects only those in the current package.

Perl4 Note #11

Page 248 When the Perl4 match operator is given an empty regex operand, it reuses the most recent successfully applied regular expression without regard to scope. In Perl5, the most recently successful *within the current dynamic scope* is reused.

Perl4 Note #3

Page 218 In Perl4, `$+` magically becomes a copy of `$&` when there are no parentheses in the regex.

Perl4 Note #4

Page 220 Perl4 interpolates `$MonthName[…]` as an array reference only if `@MonthName` is known to exist. Perl5 does it regardless.

Perl4 Note #5

Page 222 The escape of an escaped closing delimiter is *not* removed in Perl4 as it is in Perl5. It matters when the delimiter is a metacharacter. The (rather farfetched) substitution `s*2\*2*4*` would not work as expected in Perl5.

Perl4 Note #6

Page 247 Perl4 allows you to use whitespace as a match-operand delimiter. Although using newline, for example, was sometimes convenient, for the most part I'd leave this maintenance nightmare for an Obfuscated Perl contest.

Perl4 Note #7

Page 247 Remember, in Perl4, *all* escapes are passed through. (See note #5.)

Perl4 Note #8

Page 247 Perl4 does not support the four `m{…}` special-case delimiters for the match operator. It does, however, support them for the substitution operator.

Perl4 Note #9

Page 247 Perl4 supports the special `?`-match, but only in the `?…?` form. The `m?…?` form is not special.

Perl4 Note #10

Page 247 With Perl4, a call to any `reset` in the program resets all `?`-delimited matches. Perl5's `reset` affects only those in the current package.

Perl4 Note #11

Page 248 When the Perl4 match operator is given an empty regex operand, it reuses the most recent successfully applied regular expression without regard to scope. In Perl5, the most recently successful *within the current dynamic scope* is reused.

An example should make this clear. Consider:

```
    "5" =~  m/5/;        # install 5 as the default regex
    { # start a new scope . . .
       "4" =~ m/4/;     # install 4 as the default regex
    } # ... end the new scope.
    "45" =~ m//;         # use default regex to match 4 or 5, depending on which
regex is used
    print "this is Perl $&\n";
```

Perl4 prints '`this is Perl 4`', while Perl5 prints '`this is Perl 5`'.

Perl4 Note #12

Page 252 In any version, the list form of `m/…/g` returns the list of texts matched within parentheses. Perl4, however, does not set `$1` and friends in this case. Perl5 does both.

Perl4 Note #13

Page 252 List elements for non-matching parentheses of `m/…/g` are undefined in Perl5, but are simply empty strings in Perl4. Both are considered a Boolean false, but are otherwise quite different.

Perl4 Note #14

Page 253 In Perl5, modifying the target of a scalar-context `m/…/g` resets the target's `pos`. In Perl4, the `/g` position is associated with each *regex operand.* This means that modifying what you intended to use as the target data has no effect on the `/g` position (this could be either a feature or a bug depending on how you look at it). In Perl5, however, the `/g` position is associated with each *target string*, so it is reset when modified.

Perl4 Note #15

Page 255 Although Perl4's match operator does not allow balanced delimiters such as `m[…]`, Perl4's substitution does. Like Perl5, if the regex operand has balanced delimiters, the replacement operand has its own set. Unlike Perl5, however, whitespace is not allowed between the two (because whitespace is valid as a delimiter in Perl4).

Perl4 Note #16

Page 255 Oddly, in Perl4, `s'…'…'` provides a singlequotish context to the regular expression operand as you would expect, but *not* to the replacement operand—it gets the normal doublequoted-string processing.

Perl4 Note #17

Page 257 In Perl4, the replacement operand is indeed subject to singlequotish interpolation, so instances of `\'` and `\\` have their leading backslash removed before `eval` ever gets a hold of it. With Perl5, the `eval` gets everything as is.

Perl4 Note #18

Page 258 Perl5 returns an empty string when no substitutions are done. Perl4 returns the number zero (both of which are false when considered as a Boolean value).

Perl4 Note #19

Page 260 The default chunk-limit that Perl provides in

```
($filename, $size, $date) = split(…)
```

does impact the value that `@_` gets if the `?…?` form of the match operand is used. This is not an issue with Perl5 since it does not support the forced split to `@_`.

Perl4 Note #20

Page 263 Perl4's `split` supports a special match operand: if the list-context match operand uses `?…?` (but not `m?…?`), `split` fills `@_` as it does in a scalar-context. Despite current documentation to the contrary, this feature does not exist in Perl5.

Perl4 Note #21

Page 264 In Perl4, if an expression (such as a function call) results in a scalar value equal to a single space, that value is taken as the special-case single space. With Perl5, only a literal single-space string is taken as the special case.

Perl4 Note #22

Page 264 With Perl4, the default operand is `m/\s+/`, not `'•'`. The difference affects how leading whitespace is treated.

Perl4 Note #23

Page 275 With Perl4, it seems that the existence of `eval` anywhere in the program, also triggers the copy for each successful match. Bummer.

A
Online Information

In this appendix
- ***General Information***
- ***Other Web Links***

This appendix provides pointers to online information and utilities dealing with regular expressions. Since any printed listing of World Wide Web resources is in danger of quickly going out of date, I'd like to start off by pointing you to my home page, which I intend to keep current. It can be found at any of:

```
     http://enterprise.ic.gc.ca/cgi-bin/j-e/jfriedl.html

http://merlin.soc.staffs.ac.uk/cgi-bin/jfriedl/j-e/jfriedl.html
     http://linear.mv.com/cgi-bin/jfriedl/j-e/jfriedl.html
```

I can also be reached at `jfriedl@oreilly.com.` I'd like to hear your comments!

General Information

Mastering Regular Expressions

Besides keeping you abreast of the latest in regex-related URLs, you can also get the following book-related items from my home page:

- The list of errata (if any!)
- The larger or more-complex examples used throughout the book
- An online index

If you've found an error, or can't find something you think should be available, please don't hesitate to contact me.

O'Reilly & Associates

O'Reilly & Associates is a one-stop shop for computational knowledge. You can browse their online catalog at `http://www.oreilly.com/`. They have books on many of the tools I've mentioned, including Awk, Emacs, Expect, Perl, Python, and Tcl.

OAK Archive's Virtual Software Library

At `http://castor.acs.oakland.edu/cgi-bin/vsl-front`, this *huge* site holds every piece of software ever made. Well, not quite, but they have a lot. They claim "The VSL is the most powerful tool available for searching for shareware and freeware on the Internet." I don't disagree!

The *power search* option allows you to search via category, including Amiga, Atari, DOS, MacOS, Novell, OS/2, Unix, and Windows. The *archive search* allows you to search over 200 archives, including many with names ending in `-mac`, `-msdos`, `-os2`, `-unix`, and `-win95`. The *browse* option allows you to, well, browse (huge pages pre-sorted by your criteria).

The GNU Archive

The main distribution point for GNU software is:

```
ftp://prep.ai.mit.edu/pub/gnu
```

Yahoo!

Yahoo! (conversationally, just *Yahoo,* or `http://www.yahoo.com/` to your browser) is a great place to hunt down information. A good place to start is

```
http://www.yahoo.com/Computers_and_Internet/
```

and in particular, its `Software` and `Programming_Languages` sub-categories.

Other Web Links

In this section I list URLs for some specific tools. The OAK VSL and Yahoo are always default places to search, so I'll not repeat them below. For example, in this section I'll often point to source code, but binaries for DOS, MacOS, OS/2, and Windows are often available through the OAK VSL. Italics in filenames are version numbers that change from time to time; look for the latest version.

Awk

`gawk-`*`3.0.0`*`.tar.gz` at the GNU archive
Source for GNU *awk*. (Documentation in `gawk-`*`3.0.0`*`-doc.tar.gz`)

`http://netlib.bell-labs.com/cm/cs/who/bwk/`
Brian Kernighan's home page. Has source for his *One True Awk*

`http://www.mks.com/`
The MKS Toolkit has a POSIX *awk* (and many other tools) for Win95/NT

C Library Packages

`ftp://ftp.zoo.toronto.edu/pub/bookregexp.shar`
Henry Spencer's original regex package

`ftp://ftp.zoo.toronto.edu/pub/regex.shar`
Henry's latest package

`rx-`*`1.0`*`.tar.gz` at the GNU archive
GNU's `rx` package

`regex-`*`0.12`*`.tar.gz` at the GNU archive
GNU's older `regex` package

Java Regex Class

`http://www.cs.umd.edu/users/dfs/java/`
Daniel Savarese's freely usable package, providing both DFA and NFA engines (with a Perlesque regex flavor).

Egrep

`grep-`*`2.0`*`.tar.gz` at the GNU archive
Source for GNU *egrep* (and *grep*)

See the OAK VSL for many, many others.

Emacs

`emacs-`*`19.34b`*`.tar.gz` at the GNU archive
Source for GNU Emacs (Various documentation in other files beginning with `emacs-`)

`http://www.geek-girl.com/emacs/faq/`
Emacs FAQ and other information. Includes information on ports to Windows, NT, OS/2, VMS, and more.

`ftp://ftp.cs.cornell.edu/pub/parmet/`
Emacs for MacOS

Flex

`flex-`*`2.5.3`*`.tar.gz` at the GNU archive
Source and documentation for *flex*

Perl

`http://www.perl.com/perl/index.html`
The Perl Language Home Page

`http://tpj.com/`
The Perl Journal

`http://www.xs4all.nl/~jvromans/perlref.html`
Johan Vromans' *Perl Reference Guide.* You might find its published version, O'Reilly's *Perl 5 Desktop Reference,* to be more to your liking. Also, it comes pre-assembled, which can be a great time-saver if your printer doesn't print double-sided.

`http://www.wg.omron.co.jp/~jfriedl/perl/index.html`
My Perl page (some utilities I've written that might be useful)

Python

`http://www.python.org/`
The Python Language Home Page

Tcl

`http://sunscript.sun.com/`
The Tcl Language Home Page

B
Email Regex Program

Here's the optimized version of the email regex program from "Matching an Email Address" (☞ 294). It's also available online (see Appendix A).

```
# Some things for avoiding backslashitis later on.
$esc        = '\\\\';               $Period      = '\.';
$space      = '\040';               $tab         = '\t';
$OpenBR     = '\[';                 $CloseBR     = '\]';
$OpenParen  = '\(';                 $CloseParen  = '\)';
$NonASCII   = '\x80-\xff';          $ctrl        = '\000-\037';
$CRlist     = '\n\015';  # note: this should really be only \015.

# Items 19, 20, 21
$qtext = qq/[^$esc$NonASCII$CRlist"]/;               # for within "..."
$dtext = qq/[^$esc$NonASCII$CRlist$OpenBR$CloseBR]/; # for within [...]
$quoted_pair = qq< $esc [^$NonASCII] >; # an escaped character

# Items 22 and 23, comment.
# Impossible to do properly with a regex, I make do by allowing at most one level of nesting.
$ctext   = qq< [^$esc$NonASCII$CRlist()] >;

# $Cnested matches one non-nested comment.
# It is unrolled, with normal of $ctext, special of $quoted_pair.
$Cnested = qq<
   $OpenParen                            #  (
      $ctext*                            #      normal*
      (?: $quoted_pair $ctext* )*        #      special normal*)*
   $CloseParen                           #                    )
>;

# $comment allows one level of nestedparentheses
# It is unrolled, with normal of $ctext, special of ($quoted_pair  | $Cnested)
$comment = qq<
   $OpenParen                               # (
       $ctext*                              #     normal*
       (?:                                  #       (
          (?: $quoted_pair | $Cnested )  #        special
           $ctext*                       #        normal*
       )*                                #           )*
   $CloseParen                           #              )
>;
```

```
    # $X is optional whitespace/comments.
    $X = qq<
       [$space$tab]*                   #  Nab whitespace.
       (?: $comment [$space$tab]* )*   #  If commentfound, allow more
spaces.
    >;

    #  Item 10: atom
    $atom_char =
qq/[^($space)<>\@,;:".$esc$OpenBR$CloseBR$ctrl$NonASCII]/;
    $atom = qq<
      $atom char+      #  some number of atom characters . . .
      (?!$atom_char) #  . .notfolloued by something that could bepart of an atom
    >;

    #  Item 11: doublequoted string, unrolled.
    $quoted_str = qq<
        "                                       # "
           $qtext *                             #      normal
           (?: $quoted_pair $qtext * )*         #      (special normal*)*
        "                                       #                  "
    >;

    #  Item 7: word is an atom or quoted string
    $word = qq<
        (?:
           $atom                 #  Atom
           |                     #    or
           $quoted_str           #  Quoted string
        )
    >;

    #  Item 12: domain-refisjust an atom
    $domain_ref  = $atom;


    #  Item 13.: domain-literal is like a quoted string, but [. . .] instead of ". . ."
    $domainlit = qq<
        $OpenBR                               # [
        (?: $dtext | $quoted_pair )*          #      stuff
        $CloseBR                              #           ]
    >;
```

```
# Item 9. sub-domain is a domain-ref or domain-literal
$sub_domain = qq<
  (?:
    $domain_ref
    |
    $domain_lit
  )
  $X # optional trailing comments
>;

# Item 6: domain is a list of subdomains separated by dots.
$domain = qq<
     $sub_domain
     (?:
        $Period $X $sub_domain
     )*
>;
```

```
    # Item 8: a route. A bunch of "@ $domain" separated by commas, followed by a colon.
    $route = qq<
        \@ $X $domain
        (?: , $X \@ $X $domain )*   # additional domains
        :
        $X # optional trailing comments
    >;

    # Item 6: local-part is a bunch of $word separated by periods
    $local_part = qq<
        $word $X
        (?:
            $Period $X $word $X # additional words
        )*
    >;

    # Item 2: addr-spec is local@domain
    $addr_spec  = qq<
      $local_part \@ $X $domain
    >;

    # Item 4: route-addr is <route? addr-spec>
    $route_addr = qq[
        < $X                   # <
            (?: $route )?      #    optional route
            $addr_spec         #    address spec
        >                      #             >
    ];
    # Item 3: phrase........
    $phrase_ctrl = '\000-\010\012-\037';   # like ctrl, but without tab

    # Like atom-char, but without listing space, and uses phrase_ctrl.
    # Since the class is negated, this matches the same as atom-char plus space and tab
    $phrase_char =

qq/[^()<>\@,;:".$esc$OpenBR$CloseBR$NonASCII$phrase_ctrl]/;

    # We've worked it so that $word, $comment, and $quoted_str
to not consume trailing $X
    # because we take care of it manually.
    $phrase = qq<
       $word                                       # leading word
       $phrase_char *                              # "normal" atoms
and/orspaces
       (?:
           (?: $comment | $quoted_str )            # "special" comment or
quoted string
```

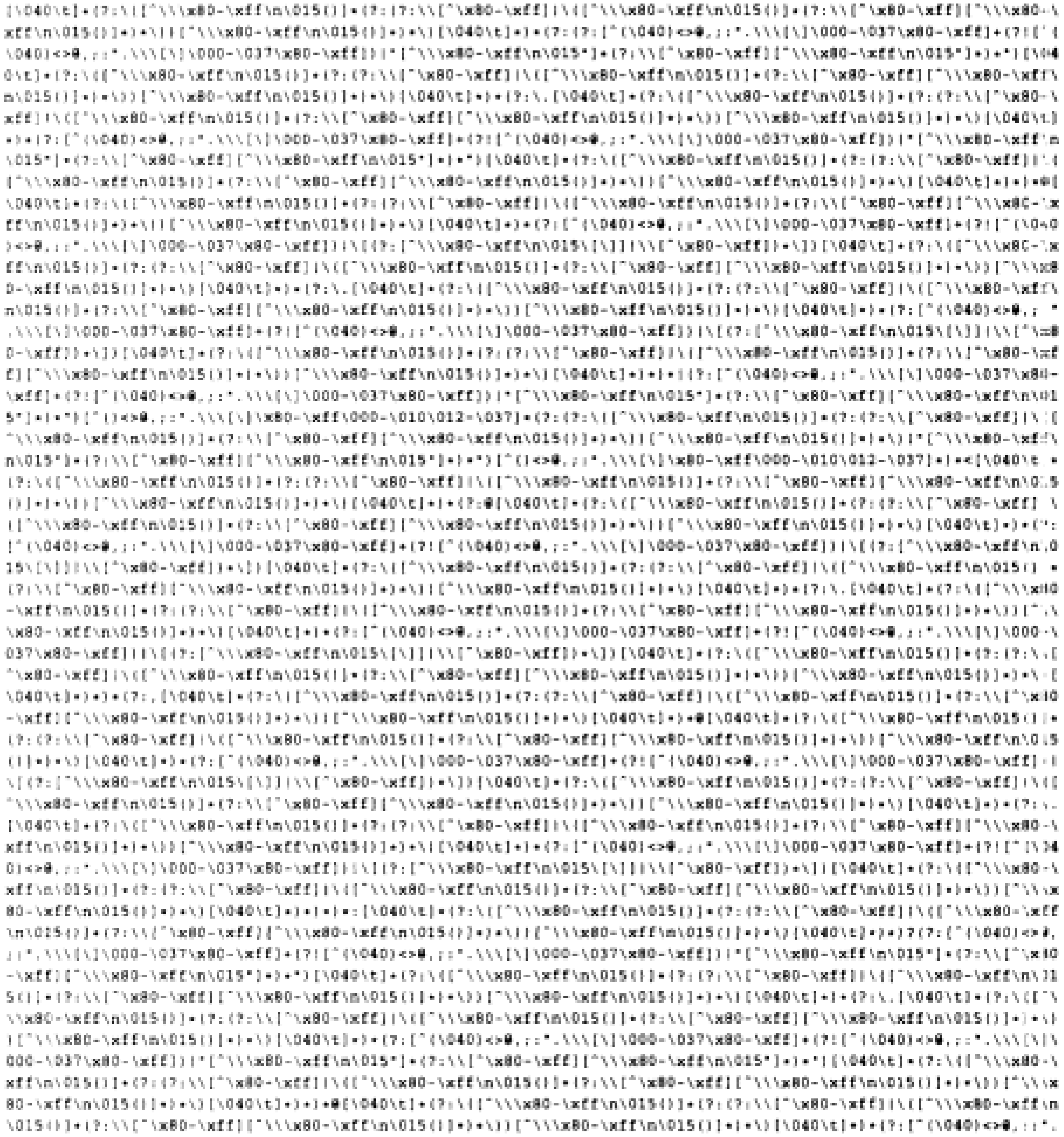
```
      $phrase_char *                 #  more "normal"
   )*
>;

## Item #1: mailbox is an addr_spec or a phrase/route_addr
$mailbox = qq<
    $X                               # optional leading comment
    (?:
           $addr_spec                # address
           |                         #  or
           $phrase  $route_addr      # name and address
     )
>;
```

After whitespace and comments are stripped with

```
$mailbox =~ s/#.*//g;
$mailbox =~ s/\s+//g;
```

(which we can do because we know there are no spaces or # in our character classes), we end up with a regex that's 6,598 bytes long. Still, it's faster than the 4,724-byte regex developed in Chapter 7. If you'd rather type in the regex directly, I've listed it here for your convenience.



```
\\\[\]\000-\037\x80-\xff]+(?![^(\040)<>@,;:".\\\[\]\000-\037\x80-\xff])|\[(?:[^\\\x80-\xff\n\015\[\]]|\\[^\x80
-\xff])*\])[\040\t]*(?:\([^\\\x80-\xff\n\015()]*(?:(?:\\[^\x80-\xff]|\([^\\\x80-\xff\n\015()]*(?:\\[^\x80-\xff
][^\\\x80-\xff\n\015()]*)*\))[^\\\x80-\xff\n\015()]*)*\)[\040\t]*)*(?:\.[\040\t]*(?:\([^\\\x80-\xff\n\015()]*(
?:(?:\\[^\x80-\xff]|\([^\\\x80-\xff\n\015()]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015()]*)*\))[^\\\x80-\xff\n\015(
)]*)*\)[\040\t]*)*(?:[^(\040)<>@,;:".\\\[\]\000-\037\x80-\xff]+(?![^(\040)<>@,;:".\\\[\]\000-\037\x80-\xff])|\
[(?:[^\\\x80-\xff\n\015\[\]]|\\[^\x80-\xff])*\])[\040\t]*(?:\([^\\\x80-\xff\n\015()]*(?:(?:\\[^\x80-\xff]|\([^
\\\x80-\xff\n\015()]*(?:\\[^\x80-\xff][^\\\x80-\xff\n\015()]*)*\))[^\\\x80-\xff\n\015()]*)*\)[\040\t]*)*)*>)
```

Enjoy!

Index

~ **operator** 187

`\1`

- in Tcl 191

`\9`

- in *awk* 185
- in Tcl 190

❖ (see questions)

✓c 78

✓ 73, 75-76, 78, 82, 182, 184

▲ 73, 75

✚ 73, 78

✘ 73

NL xx, 76, 174, 182-183, 194, 282

- (see also `\n` (newline))

● xx

TAB xx, 40, 42, 187, 197

(see also `\t` (tab))

`\(…\)` (see parentheses)

`\\\\` 76, 133, 221, 296

`\+` (see plus)

`\|` (see alternation)

`\>` (see word anchor)

`\` (see escape)

`\*` (see star)

`\?` (see question mark)

`\<` (see word anchor)

`\1` (see backreference; octal escapes)

in Perl 36, 218-219, 227, 243, 246, 272, 276, 279, 284, 305

`\9` 74, 83

`\A` 236

`\a` (alert) 72, 241

`\B` 240-241, 297

`\b` 72

dual meaning in Emacs 76

  dual meaning in Python 76
  in Perl 41, 76, 222, 240-241, 297
\d (digit) 77, 228, 241, 287
\D (non-digit) 77, 241
\E 221, 240, 245
\e (escape) 72, 241
\f (form feed) 72, 241
\G 217, 236-240, 251, 292
\l 245
\L 245-246
\n(newline) 72, 233, 235, 241, 296
␤ xx
  machine-dependency 72
  uncertainty in Tcl 189
\Q 221, 240, 245
\r (carriage return) 72, 241
  machine-dependency 72
  uncertainty in Tcl 189
\S (non-whitespace) 77, 241
\s (whitespace) 77, 241, 287

introduction 42

equivalence to character class 57

and POSIX 242

\s*char* (whitespace syntax class) 78

\t (tab) 72

  TAB xx

\v (vertical tab) 72, 241

\w (non-word) 77, 230, 240-241

  and POSIX 242

\w (word Constituent) 77, 230, 240-241

  in Emacs 78

  many different interpretations 63

  and POSIX locale 65

\x (see hexadecimal escapes)

s/…/…/ (see Perl, substitution)

m/…/ (see Perl, match)

/…/

  in Perl (see Perl, match)

/e (see Perl, modifier: evaluate replacement (`/e`))

/eieio 258

/g (see Perl, modifier: global match (`/g`))

/i (see Perl, modifier: case insensitive (`/i`))

/m (see Perl, modifier: multi-line mode (`/m`))

/o (see Perl, modifier: cache compile (`/o`))

/osmosis 249

/s (see Perl, modifier: single-line mode (`/s`))

/x (see Perl, modifier: free-form (`/x`))

`-  -` 191

-all 68, 173, 175, 191

-DDEBUGGING 155, 285

-indices 135, 191, 304

-nocase 68, 190-191

<> 49, 232, 234, 284, 289, 293

  and `$_` 55

$' (see Perl, $&, `$'`, and `` $` ``)

(…) (see parentheses)

$+ (see Perl, `$+`)

[=   =] (see POSIX)

[.….] (see POSIX)

| (see alternation)

^ (see caret; character class, negated; line anchor)

.*

  warning about 51

$ (see dollar; line anchor)

(?!...) (see lookahead, negative)

$` (seePerl, $&, $',and $`)

$$ 257

$& (see Perl, $&, $', and $`)

=~ 34, 220, 248, 250, 262

?…? 247, 251, 254-255, 308

  in Perl4 306, 308

* (see star)

[:…:] (see POSIX)

[^…] (see character class, negated)

(?=…) (see lookahead)

$* 234-235, 254

  and quieting warnings 235

!~

  in *awk* 187

  in Perl 39, 251

. (see dot)

? (see question mark)

[…**]** (see character class)

{…**,**…**}** (see interval)

$_ 55, 236, 250, 260, 272, 282

  bug with study 289

$/ 31, 54, 232, 234, 293

+ (see plus)

{…**} (for interpolation)** 222

$0 218

$1 (see Perl)

/…**/**

  in *awk* 187

A

accidental feature of Perl 258

Adobe Systems xxiii

Aho, Alfred 62, 90, 104, 183

alert (see `\a` (alert))

alphanumeric sequence 14

alternation 12, 84
  and backtracking 148
  and building a regex in a string 303
  vs. character class 13, 115
  efficiency 140-142, 148, 155, 164, 177, 179, 272, 291, 300
  faux 113
  greediness 112-114
  importance of order 112, 130, 141-143, 179
    for correctness 114, 125, 130
    for efficiency 131, 142
  non-greedy more powerful 114
  and parentheses 12

Jump

analogy

backtracking

bread crumbs 102-103

stacking dishes 103

ball rolling 162

"between" xvii

car

chrome and finish 70

DFA and NFA 88

funny car 85

gas additive 93

learning to build 28

learning to drive 28

main engine description 87-88

muscle car 85

POSIX standard 89

supercharging 155

transmission 92, 146, 151, 154, 158, 178 charging batteries 118

chicken and egg 204

cooking regex operands 202

creative juices 102, 108, 113

dirty ball rolling 162

Dust Buster 118

engine (see analogy, car)

first come, first served 96, 98

learning regexes

- building a car 28
- driving a car 28
- Pascal 32
- playing rummy 30

regex

- dialect 21
- as filename patterns 4
- flavor 21
- as a language 5, 25

regex-directed match 99

transparencies (Perl's local) 214

anchor (see line anchor; word anchor)

ANSI escape sequences 56

any

  alternative (see alternation)

  character (see dot)

  number of matches (see quantifier, star)

青山健治 xxiii

Aoyama, Kenji xxiii

array context (see Perl, context)

ASCII 26, 65, 73, 242

  carriage return and linefeed 72, 296

Asian character encoding 26

assumptions 11, 53, 56, 125, 128, 132, 168, 172, 236

asterisk (see star)

AT&T Bell Labs 61, 184-185

`AutoLoader` module 278

`AutoSplit` module 278

awk 183-188

  ~ operator 187

  !~ operator 187

`\9` 185

`/.../` 187

AT&T 74, 185

character class

  escapes within 186

  inverted range 186

empty regex 186

engine type 90

flavor chart 184

GNU 90, 185

  and alternation 119

  chart of shorthands 73

  `gensub` 121, 187

  `--posix` 185

  `--re-interval` 185

  source 311

`gsub` 68, 187

hexadecimal escapes 185

history 183

input-field separator 187

line anchors 82

mawk 90, 161, 185

    optimizations 157, 161

MKS 185, 311

Mortice Kern Systems 185

`\n`, *vs. egrep* 184

nawk 185

and null character 186

oawk 185, 311

    source 311

octal escapes 185

One True Awk 185, 311

    source 311

record separator in Perl 263

regex delimiter 187

source 311

☐ Jump

awk (cont'd)

- `split` 187-188
- strings as regexes 187
- `sub` 68, 187
- `\t`, *vs. egrep* 184

B

backreferences

- introduced (with *egrep)* 19-20
- and DFAs 93, 120-121
- in Perl 243
  - and character class 227, 243
- remembering text 19

backtracking 102-107

- and alternation 148
- computing backtracks 144
- global view 145
- LIFO 103
- option selection 103

    POSIX NFA example 147

    pseudo-backtrack 106

    saved states 104-108

    "unmatching" 107, 109

bar (see alternation)

Barwise, J. 60

`Basename` module 278

Basic Regular Expressions 64

Beard, Paul xxiii

benchmark 144-145, 151-154, 156, 168, 175, 179, 198, 229, 267, 276-277, 279-280, 303

    in Emacs 197

    in Perl 284-285

    `time-now` 197

`Benchmark` module 278

Berke, Wayne xxii, 243

Berkeley 62

`BigFloat` module 278

Biggar, Mark xxii

blanks (see whitespace)

boring DFA 101

Boyer-Moore 155

brace (see interval)

bracket (see character class)

bracket expressions 79

BRE (see POSIX, Basic Regular Expressions)

bread-crumb analogy 102-103

Brennan, Michael 90, 157, 185

`Build_MatchMany_Function` 271, 273

Bulletin of Math. Biophysics 60

byte (see character)

- any (see dot)

C

C language

- library support for POSIX locales 65
- removing comments 173, 292-293
- unrolling comments 171-176

camels, dead flea-bitten carcasses 208

`Cap` module 278

capitalization

  - ignoring differences (see case-insensitive match)

capturing text (see parentheses; parentheses; backreferences)

caret (see character class, negated)
  - line anchor 8, 81
  - and Perl line mode 232-235

`Carp` module 277-278

carriage return (see character, encoding, ASCII; `\r` (carriage return))

`case-fold-search` 193

case-insensitive match 19
  - bugs 182
  - and dot 280
  - with *egrep* 19
  - in Emacs 193
  - *grep* history 280
  - implementation of 278
  - in the Latin-1 encoding 244
  - in Perl (see Perl, modifier: `/i`)
  - and POSIX locale 65
  - in Python 70
  - in Tcl 68, 190-191

cast 211

Celsius to Fahrenheit conversion (see temperature conversion)

Chachich, Mike xxiii

character

  any (see dot)

  chart of shorthands 73

  encoding

    ASCII 26, 65, 72-73, 242, 296

    EBCDIC 26

    EUC-JP 26

Jump

character, encoding (cont'd)

        ISO-2022-JP 26

        JIS 26

        Latin-1 26, 65, 70, 244

        Unicode 26

    machine-dependent codes 72

    null 78, 186, 190

    as opposed to byte 26, 243

    POSIX equivalents 80

    shorthands 77

characterclass 9, 78

    vs. alternation 13, 115

    chart of shorthands 73

    escapes within 79, 131

        in *awk* 186

    inverted range

        in *awk* 186

    mechanics of matching 93

metacharacters 79

negated 10, 79

    must match character 11

    must still match 12, 28, 171

    and newline 79

    vs. non-greedy constructs 226-227

in Perl 243-245, 279

    and `\n` 235

    and `/i` 280

    and `/x` 223, 231, 295, 303

    and backreferences 227, 243

    instead of dot 226

    vs. negative lookahead 228

    shorthands 73, 241

positive assertion 79

of POSIX bracket expression 80

range 9, 79

    inverted 186

as a separate language 9, 41

shorthands 77

`CheckNaughtiness` 279

checkpoint (see unrolling the loop)

Chinese text processing 26

Christiansen, Tom xxii, 208

circumflex (see caret)

class (see character class)

coerce 211

`Collate` module 242, 245

collating sequences 65, 81

command shell 7

COMMAND.COM 7

comments

  in an email address 297

  matching of C comments 168-176

  matching of Pascal comments 165

  in a regex 230-231

  within a regex 230-231

common patterns 163

Communications of the ACM 60

comparison of engine types (see DFA)

- compile caching 192
  - discussed 158
  - in Emacs 160, 197
  - in Perl 224, 268-273
  - in Tcl 160
- *Compilers-Principles, Techniques, and Tools* 104
- **`Config`** module 216
- Confusion
  - dot *vs.* character class 79
- confusion
  - ^ in Python 58
  - ⌈`[•TAB]*`⌋ *vs.* ⌈`(•*|TAB*)`⌋ 40
  - =~ *vs.* == *vs.* = 34
  - balanced parentheses 126-127
  - character class and alternation 13
  - continuation lines 122
  - dot *vs.* character class
    - Perl `/s` modifier 226
  - metacharacter contexts 41

  - non-greedy *vs.* negated character class 227
  - Perl's `print` and floating point 35
  - unexpected text 127, 131
  - word *vs.* alphanumeric sequence 14
  - ⌜`X{0,0}`⌟ 84
  - `\x0x`*ddd*… 190
- Constable, Robert xxii, 60
- context
  - of metacharacters (see metacharacter, multiple contexts)
- continuation lines 116, 122
- control (see regex-directed matching)
- conversion
  - of data types 211
  - temperature (see temperature conversion)
- **`Copy`** module 278

☐ Jump

Couch, Norris xxiii

counting parentheses 19, 44, 97, 229

`cperl-mode` 194, 243

creative juices (see NFA, creative juices)

Cruise, Tom 45

crumb analogy 102-103

CSV text 204-208, 227, 231, 290

cultural support (see POSIX locale)

curley braces (see interval)

D

dash (see character class, range)

`DB_File` module 278

default regex 248, 269

  in Perl4 306

  with split 264

`define-key` 69

delimited text 129-131

  standard formula 129, 169

delimiter (see regex delimiter)

Delphi 120

`describe-syntax` 194

Deterministic Finite Automaton (see DFA)

DFA 101

- first introduced 88
- compared with NFA 100, 118-121, 142, 145, 160-162
- efficiency 118
- fast, consistent, and boring 101
- first-character discrimination 154
- implementation ease 120
- lazy evaluation 119
- no backreferences 93, 120-121
- testing for 145, 160-162
- in theory, same as an NFA 104

`diagnostics` module 278

`DirHandle` module 278

dish-stacking analogy 103

dollar

- line anchor 8, 82

  - matching value (with cents) 23
  - and Perl line mode 232-235
  - for Perl variable 33
  - sometimes variable interpolation 273
- DOS 7
- dot 11, 78
  - and case-insensitive matching 280
  - mechanics of matching 93
  - and newline 131, 158
  - Perl 226
  - and Perl line mode 232-235
  - POSIX 78
- **`dotsh.pl`** package 278
- doubled-word example
  - description 1
  - solution in *egrep* 20
  - solution in Emacs 69
  - solution in *grep* 61
  - solution in Perl 31, 54
  - solution in Python 57

doublequoted string

  allowing escaped quotes 129

  matching 22, 205, 221, 293

  matching in Emacs 76

  Perl 241, 294

    processing 221

    variables appearing within 33, 219, 305

  sobering example 140-145

Dougherty, Dale 8

dragon book 104

`dumpvar.pl` package 278

Dust Buster 118

`DynaLoader` module 278

dynamic scope 211-217

E

EBCDIC 26, 73

ed xxii, 61

efficiency 107, 150-153, 265-289

  and alternation 140-142, 148, 155, 164, 177, 179, 272, 291, 300 and backtracking 118

non-greedy quantifiers 152, 156

of quantifiers 143, 155-156, 226

of DFA 118

of POSIX NFA 118

parentheses 107

and parentheses 276, 282

penalty 273-282

in Perl 265-289

    and parentheses 276, 282

    penalty 273-282

    substitution 281-284

regex-directed matching 118

Jump

efficiency (cont'd)

  substitution 281-284

  text-directed matching 118

egrep 7

  case-insensitive match 19

  engine type 90

  flavor summary 29

  GNU 120

    and backreferences 94

    chart of shorthands 73

  history 62, 183

  `\n`, *vs. awk* 184

  only care whether a match 23

  source 311

  strips end-of-line character 12

  `\t`, *vs. awk* 184

Emacs 90, 192-198

  `\b`, dual meaning of 76

benchmarking 197

`case-fold-search` 193

case-insensitive matching 193

chart of shorthands 73

`cperl-mode` 194, 243

`define-key` 69

`describe-syntax` 194

for DOS 312

doublequoted string matching 76

elisp 76, 192

escape is dropped string processing 76

FAQ 312

first Unix version xxii

first-character discrimination 178

future version 198

`isearch-forward` 192

`isearch-forward-regexp` 161, 192

line anchors 82

`looking-at` 196

  - for MacOs 312
  - `match-data` 196-197, 304
  - `match-string` 196-197
  - optimization
    - compile caching 160, 197
    - first-character discrimination 154, 179, 197
  - and Perl history 208
  - support for POSIX locale 66
  - `posix-search-forward` 161
  - `regexp-quote` 193
  - `replace-match` 196-197
  - `re-search-forward` 68-69
  - source 312
  - strings as regexes 69, 75
  - superficial flavor chart 194
  - syntax class 78
    - and `isearch-forward` 192
    - summary chart 195
  - `time-now` 197
- email

- generating table of contents 3, 7
  - problem 3
  - solution 7
- matching an address 294-304, 313

Embodiments of Mind 60

empty regex
- in *awk* 186
- in Perl 259

encoding (see character, encoding)

end
- of line 8, 82

end, of word (see word anchor)

engine 24
- hybrid 121
- implementation ease 120
- listing of programs and types 90
- testing type 145, 160-162

`English` module 277-278

English *vs.* regex 171

ERE (see POSIX, Extended Regular Expressions)

- errata (for this book) 309
- escape 21
  - term defined 25
  - unescapable 230
  - within character class 79, 131, 186
    - in *awk* 186
- *escape is dropped* string processing 76
- *escape is passed* string processing 77
- EUC-JP encoding 26
- **`eval`** 222, 235, 256, 279, 284, 308
  - and `/o` 270
  - string *vs.* block 271
- example
  - ⌈`[•TAB]*`⌋ *vs.* ⌈`(•*|TAB*)`⌋ 40
  - ⌈`(?!0+\.0+\.0+\.0+$)`⌋ 125
  - ⌈`07[-./]04[-./]76`⌋ 11

☐ Jump

example (cont'd)

⌈Ù.*([0-9]+)⌋ 98

⌈^.*[0-9][0-9])⌋ 97, 107

⌈\$[0-9]+(\.[0-9)[0-9])?⌋ 23

9.0500000037272 46, 110

getting from the Web 309

⌈"[^"]*"⌋ 22

⌈"([^"\\]|\\.)*"⌋ 168, 205

⌈"(\\.|[^"\\])*"⌋ 76, 131, 140, 162

⌈".*"!⌋ 147

⌈".*"⌋ 108, 125, 145, 150, 155

⌈^.*(.*)⌋ 98

⌈"([^"]|\\")*"⌋ 130

⌈"\"\"\""⌋ 163

⌈^(.*)/(.*)$⌋ 135

⌈[aa-a](b|b{1}|b)c⌋ 101

⌈`a*((ab)*|b*)`⌋ 113

⌈`[a-zA-Z_][a-zA-Z_0-9]*`⌋ 22

⌈`<B>.*</B>`⌋ 109

C functions 126-127

commaizing a number 291

⌈`(\.\d\d[l-9]?)\d*`⌋ 47, 110, 113

dynamic scope 214

email
  - generating table of contents 3, 7
  - matching an address 294-304, 313

⌈`fat|cat|belly|your`⌋ 92

filename and path 132-136

finding doubled words
  - description 1
  - solution in *egrep* 20
  - solution in Emacs 69
  - solution in *grep* 61
  - solution in Perl 31, 54
  - solution in Python 57

floating-point number 127

form letter 45-46

⌈^(From|Subject):•⌋ 3, 7

⌈^(From|Subject|Date):•⌋ 13

⌈gr[ea]y⌋ 9

⌈<H[123456]>⌋ 9

'hot•tonic•tonight!' 102

⌈<•*HR(•+SIZE•*=•*[0-9]+)?•*>⌋ 20

⌈<•*HR•+SIZE•*=•*[0-9]+•*>⌋ 18

HTML 1, 9, 17-18, 69, 109, 127, 129, 229, 264

indicates 92

IP 5, 123-125, 167, 266

⌈Jan•[0123][0-9]⌋ 113

'Jeffreyrey•Friedl' 45

⌈July?•(fourth|4(th)?)⌋ 16

mail processing 48-54

makudonarudo 108, 142-145, 148, 164

'megawatt•computing' 20

'M.I.T.' 83

⌜[.\n]⌟ 131

'NE14AD8' 83

⌜normal*(special normal*)*⌟ 164-165, 301

⌜ora\2Ecom⌟ 185

⌜ora**x2E**com⌟ 74, 185

Perl

- Perl 214
- scope 214

prepending filename to line 56

removing whitespace 282, 290-291

'ResetSize' 32

scope 214

⌜(self)?(selfsufficient)?⌟ 116-117

sobering 140-145

stock pricing 110

⌜^Subject:•⌟ 95

'<SUCKER>' 45

temperature conversion 33-43, 199, 281

  - ⌈(this|that|)⌋ 186
  - time of day 23
  - ⌈to(nite|knight|night)⌋ 99-100, 102
  - ⌈topological⌋ 117
  - ⌈to(ur(nament)?)??⌋ 112
  - ⌈tour|to|tournament⌋ 112
  - ⌈[Tt]ubby⌋ 159
  - using $+ 290
  - variable names 22
  - ⌈^\w+=.*(\\\n.*)*⌋ 122
  - \x0x*ddd*… 190
  - ⌈X(.+)+X⌋ 161
  - ⌈/x**(**[^x]**|**x+[^/x]**)***x+/⌋ 170
  - ZIP codes 236-240
- Expect 90
- *expr* 90
- Extended Regular Expressions 64
- **ExtUtils::Install** module 278

ExtUtils::Liblist module 278

ExtUtils::MakeMaker module 278

ExtUtils::Manifest module 278

`ExtUtils::Mkbootatrap` module 278

`ExtUtils::mksymlists` module 278

`ExtUtils::MM_nix` module 278

`ExtUtils::testlib` module 278

F

Fahrenheit to Celsius conversion (see temperature conversion)

faux alternation 113

`Fcntl` module 278

de Fermat, Pierre 86

fetching URLs 73

file globs 4

`File::Basename` module 278

`FileCache` module 278

`File::Copy` module 278

`File::Find` module 278

`FileHandle` module 278

filename

  isolating 132-136

- patterns (*globs*) 4
- prepending to line 56

`File::Path` module 278

find 90

`Find` module 278

first-character discrimination 151, 179, 192, 197
- discussed 154
- in Emacs 154, 179, 197
- in Perl 286

first-class metacharacter 62, 82, 84

fixed-string check 178-179, 192
- discussed 155
- in Perl 280, 286, 288

flavor 21
- changing 69
- changing on the fly (see Python, flavor)
- hidden differences 184
- Perl 225-246
- Python resembling Emacs 69
- questions you should be asking 63, 181, 185

  - superficial chart
    - *awk* 184
    - *egrep* 29
    - Emacs 194
    - general 63, 182
    - *grep* 183
    - Perl 201, 203, 225, 232, 236, 241, 245
    - Posix 64
    - Tcl 189
  - variety 63, 181-183, 185
- *flex* xxii, 74, 90
  - chart of shorthands 73
  - engine type 90
  - source 312
- floating-point number 127
- **`foreach`** *vs.* **`while`** *vs.* **`if`** 256
- foreign language support (see POSIX, locale)
- form letter example 45-46
- **`format`** 35
- frame of mind 6

freeflowing regex 173-174

free-form regex (see Perl, modifier: freeform `(/x)`)

Friedl

  Alfred 114

  brothers 30

  Jeffrey (home page) 309

  Liz 30

  parents xxi

function *vs.* other 159

G

`GDBM_File` module 278

`Getopt::Long` module 278

global

  vs. local (greediness) 120

  match (see Perl, modifier: global match `(/g)`)

  vs. private (variable) 211-216

  variable (see Perl, variable)

globs 4

GNU

awk 90, 185
  - and alternation 119
  - chart of shorthands 73
  - `gensub` 121, 187
  - `--posix` 185
  - `--re-interval` 185
  - source 311

☐ Jump

GNU (cont'd)

  egrep 120

    and backreferences 94

    chart of shorthands 73

  Emacs (see Emacs)

  find 90

  grep 90, 181

    shortest-leftmost match 121, 157

    source 311

  software archive 310

Gosling James xxii, 208

greediness 94

  and alternation 112-114

  deference to an overall match 98, 111, 131, 170, 226, 230

  first come, first served 96, 98

  local *vs.* global 120

  non-greedy efficiency 152, 156

  regex-directed 106

- too greedy 97

green dragon 104

grep 90

- as an acronym 61
- case-insensitive match 280
- engine type 90
- flavor chart 183
- GNU 90, 181
  - shortest-leftmost match 121, 157
  - source 311
- history 60
- regex flavor 61
- SCO 182
- source 311
- variety of flavors 181
- `-y` option 62

`grep` (Perl operator) 239

grouping (see parentheses)

`gsub`

- in Python 58

  - in *awk* 68, 187

H

- Haertel, Mike 90
- Haight, Dick 90
- Halpern Jack xxi-xxiii, 3
- **`Harness`** module 278
- 春遍雀來 xxi-xxiii, 3
- **`Hash`** module 278
- hexadecimal escapes 74
  - in *awk* 185
  - in Perl 241, 243-244, 257
  - inTcl 190
- Hietaniemi, Jarkko 242
- highlighting
  - with ANSI escape sequences 56
- history
  - AT&T Bell Labs 61, 184
  - *awk* 183
  - Berkeley 62

- edtrivia 61
- *egrep* 62, 183
- *grep* case-insensitive match 280
- of Perl 208, 258
- of regexes 60-62

horizontal rule 17

`Hostname` module 278

HTML 1, 9, 17-18, 69, 109, 127, 129, 229, 264

HTTP and newlines 73

hyphen (see character class, range)

I

`I18N::Collate` module 242, 245

IBM

- 370 and EBCDIC 73
- `7094` object code 61
- ThinkPad xxiii, 144

ice trays xxiv

identifier

- matching 22

`if`

  - shortest branch first 39
  - *vs.* `while` *vs.* `foreach` 256

ignoring case (see case-insensitive match)

implementation
  - case-insensitive match 19, 65, 68, 70, 182, 190-191, 193, 244, 278, 280
  - match engine 120

`implicit` 287

implicit line anchor optimization 133, 192
  - discussed 158
  - in Perl 287

incontinent 104

input recordseparator 31, 54, 232, 234, 293

`Install` module 278

☐ Jump

internationalization 242

interpolation

  in Perl 33, 219, 232, 246, 248, 255, 266, 283, 295, 304-305, 308

interval 18, 84

  non-greedy 83

  ⌈`x{0,0}`⌋ 84

inverted character class (see character class, negated)

IP

  address 5, 123-125, 266

  matching a hostname 167

`IPC::Open2` module 278

`IPC::Open3` module 278

Iraq 11

`isearch-forward` 192

`isearch-forward-regexp` 161, 192

ISO-2022-JP encoding 26

ISO-8859-1 encoding 26, 244

J

Japanese
    character encoding 26
    fonts xxiii
    正規表現は簡単だよ! 5
    text processing 26
Java 311
JIS encoding 26
Joy, Bill 90
Just Another Perl Hacker 258

K

keeping in synch 173, 206, 236-237, 290
Keisler, H. J. 60
Kernighan, Brian xxii, 90, 183, 185, 311
Kleene, Stephen 60
The Kleene Symposium 60
Kleinedler, Steve xxii
小林劍 xxi-xxiii, 26
Korean text processing 26
Kunen, K. 60

L

labeling parentheses 97

language

  analogy 5, 25

  character class as a 13

  identifiers 22

  support (see POSIX, locale)

Latin-1 encoding 26, 65, 70, 244

lazy

  evaluation 119

  matching 83

  quantifier (see quantifier, non-greedy)

lazy evaluation 119

`lc` 44, 245

`lcfirst` 245

length cognizance optimization 197

  discussed 157

Lesk, Mike xxii, 90

less 90

letter (see character)

    ignoring case (see case-insensitive match)

    to Tom Cruise 46

Lewine, Donald 243

lex xxii, 61, 90

    flex xxii, 73-74, 90, 312

    history 183

    line anchors 82, 84

    and trailing context 120

lex & yacc 8

`lib` module 278

Libes, Don xxii, 89-90

`Liblist` module 278

Life with Unix xxii, 89

LIFO backtracking 103

limiting scope 16

line

    egrep strips end 12

    vs. string 50, 81-83

line anchor 58, 82, 232-235, 287

    caret 8, 81

  - chart 82, 232, 236
  - dollar 8, 82
  - in *lex* 84
  - as lookbehind 230
  - mechanics of matching 93
  - and newline 158
  - optimization 92, 119, 158, 232, 287
  - Python
    - surprises 58
  - variety of implementations 62, 84, 181
- line mode (see Perl, line mode)
- linefeed (see character, encoding, ASCII)

list context 210

literal text 5

  match mechanics 93

  Perl optimization 286, 288

`local` 212, 273-274

local failure

  and backtracking 105

local greediness *vs.* global 120

`Local` module 278

locale (see POSIX)

"A logical calculus of the ideas immanent in nervous activity" 60

`Long` module 278

longest-leftmost match 91, 115, 117, 119, 184, 192

  and POSIX 116

  shortest-leftmost 121, 157

lookahead 228-230

  and $& 228

  and DFAs 120

  - negative 125, 171
    - *vs.* character class 228
    - and parentheses 229
- lookbehind 130, 171, 229, 240
  - with an anchor 230
- `looking-at` 196
- Lord,Tom 120
- Lunde, Ken xxi-xxiii, 26

M

- `m/`...`/` (see Perl, match; Perl, match)
- machine-dependent character codes 72
- MacOS 72, 189
  - Emacs 312
- maintenance nightmare 217, 228, 240, 306
- `MakeMaker` module 278
- `Manifest` module 278
- *Mastering Regular Expressions* 1-342
  - online information 309
- 増田清 xxiii

Masuda, Keith xxiii

match (see Perl)

  assumptions 11, 53, 56, 125, 128, 132, 168, 172, 236

  case-insensitive 19

    bugs 182

    and dot 280

    with *egrep* 19

    in Emacs 193

    grep history 280

    implementation of 278

    in the Latin-1 encoding 244

    and POSIX locale 65

    in Python 70

    in Tcl 68, 190-191

  computing backtracks 144

  DFA *vs.* NFA S, 142

  efficiency 107, 118, 150-153, 265-289

    and alternation 140-142, 148, 155, 164, 177, 179, 272, 291, 300

    and backtracking 118

    non-greedy quantifiers 152, 156

  of quantifiers 143, 155-156, 226

  of DFA 118

  of POSIX NFA 118

  parentheses 107

  and parentheses 276, 282

  penalty 273-282

  in Perl 265-284

  regex-directed matching 118

  substitution 281-284

  text-directed matching 118

extent

  don't care with *egrep* 23

in a string 24

keeping in synch 173, 206, 236-237, 290

mechanics

  anchors 93

  character classes and dot 93

  literal text 93

  simple parentheses 93

minimal 83

neverending 144, 160, 162, 175, 296, 299, 301

  avoiding 164-166

non-greedy (see quantifier, non-greedy)

in Perl (see Perl, match)

regex-directed 99, 101, 108, 139

  alternation 122

  and backreferences 219

  control benefits 99

  efficiency 118

  essence 102

  and greediness 106

match (cont'd)
  - side effects 179
    - intertwined 39
    - in Perl 36, 251, 263
    - using 135
  - speed 118
  - text-directed 99-100
    - efficiency 118
    - regex appearance 101, 106

match cognizance optimization 157, 162

`match-data` 196-197, 304

matching
  - balanced parentheses 126-127
  - C comments 171-176
  - continuation lines 116, 122
  - delimited text 129-131
    - standard formula 129, 169
  - dollar amount 23

    - doublequoted string
        - in Emacs 76
    - email address
        - RFC 822 294
    - an email address 294-304, 313
    - floating-point number 127
    - HTML 1, 9, 17-18, 69, 109, 127, 129, 229, 264
    - IP
        - address 5, 123-125, 266
        - matching a hostname 167
    - program identifier 22
    - time of day 23
    - not on a word basis 7
- matching dollar value (with cents) 23
- **`match-string`** 196-197
- **`Math::BigFloat`** module 278
- Maton, William F xxii, 32, 309
- *mawk* 90, 161, 185
    - optimizations 157, 161
- McCulloch, Warren 60

McMahon, Lee 90

metacharacter 5

  in a character class 79

  conflict between string and regex 273

  differing contexts 10

  first class 62, 82, 84

  multiple contexts 41

metasequence 25

MIFES 119

minimal matching 83

`minlen` 155

`Mkbootstrap` module 278

MKS 75, 90, 161, 185, 311

`Mksymlists` module 278

`MM_Unix` module 278

`MM_VMS` module 278

mode (see Perl, modifier: single-line mode`(/s)`;

  Perl, modifier: multi-line mode`(/m)`;

  Perl, `cperl-mode`;

  Perl, file slurp mode;

  Perl, line mode)

modifier (see Perl)

more 90

Mortice Kern Systems 75, 90, 161, 185

motto (of Perl) 201, 266-267

Mulder, Hans 258

multi-line mode (see Perl, modifier: multiline mode (`/m`))

multi-match anchor (**`\G`**) 217, 236-240, 292

  not the match start 251

multiple-byte character encoding 26

`must have` 286, 288

`my` 211-212, 273-274

N

named subexpressions 228, 305

naughty Perl modules 278

Navigator 17

nawk 185

need cognizance optimization 161

  discussed 157

needless small quantifiers 156

negated character class (see character class)

negative lookahead 125, 171

  vs. character class 228

  and parentheses 229

Nerode, Anil xxii

nervous system

  and regular expressions 60

nested parentheses 16, 40, 97

  example 51

Netscape 17

neurophysiologists

  early regex study 60

neverending match 144, 160, 162, 175, 296, 299, 301

  avoiding 164-166

`newgetopt.pl` package 278

newline

  and dot 131, 158

  and HTTP 73

  `\n` xx, 72, 189, 233, 235, 241, 296

  and negated character class 79

  and Perl line mode 226, 231-235, 249, 253

  stripped by *egrep* 12

NFA 101

  first introduced 88

  and alternation 112

  compared with DFA 100, 118-121, 142, 145, 160-162

  creative juices 102, 108, 113

  essence 102

  freeflowing regex 173-174

  implementation ease 120

  nondeterminism 144, 166

    checkpoint 164-166

- testing for 145, 160-162
- theory 104

nightmare (of the maintenance variety) 217, 228, 240, 306

日本語情報処理 26

nm xxii

nomenclature 24

non-capturing parentheses 152, 227

Nondeterministic Finite Automaton (see NFA)

non-greedy
- alternation 114
- *vs.* negated character class 226-227
- quantifier 83, 110, 206, 225
  - alternation pitfalls 113
  - efficiency 226
  - interval 83
  - *vs.* negated character class 226
  - in Perl 83, 291
  - plus 83
  - question mark 83
  - star 83

nonregular sets 104

normal (see unrolling the loop)

Nudelman, Mark 90

null character

  and *awk* 186

  and dot 78

null character (cont'd)

  and POSIX 78

  and Tcl 190

O

OAK Archive 310

oawk 185

  source 311

Obfuscated Perl Contest 248, 258, 306

objects

  regexes as 69

octal escapes 74

  in *awk* 185

  in Perl 222, 241, 243, 295

    vs. backreferences 243

  - in Tcl 190
- **`ODM_File`** module 278
- Omron Corporation xxiii, 309
- オムロン株式会社 xxiii, 309
- one or more (see quantifier, plus)
- One True Awk 185
  - source 311
- **`Open2`** module 278
- **`open2.pl`** package 278
- **`Open3`** module 278
- **`open3.pl`** package 278
- optimization 154-160
  - Boyer-Moore 155
  - compile caching 192
    - discussed 158
    - in Emacs 160, 197
    - in Perl 224, 268-273
    - in Tcl 160
  - disabled by parentheses 156

first-character discrimination 151, 179, 192, 197
  discussed 154
  in Emacs 154, 179, 197
  in Perl 286
fixed-string check 178-179, 192
  discussed 155
  in Perl 280, 286, 288
`\G` 238
implicit line anchor 133, 192
  discussed 158
  in Perl 287
lazy evaluation 119

optimization (cont'd)
  length cognizance 197
    discussed 157
  match cognizance 157, 162
  need cognizance 161
    discussed 157
  needless small quantifiers 156
  of Perl's substitution 281-284
  in Python 152
  simple repetition 192, 197
    discussed 155
  string/line anchors 92, 119, 232, 287
    discussed 158
  twists and turns 177
option selection (see backtracking)
optional (see quantifier, question mark)
or (see alternation)
Oram, Andy xxi-xxii, 5

- O'Reilly & Associates
  - *lex & yacc* 8
  - *Mastering Regular Expressions* 1-342
    - online information 309
  - 日本語情報処理 26
  - *Perl 5 Desktop Reference* 312
  - *POSIX Programmer's Guide* 243
  - *Programming Perl* 202-203
  - *Understanding CJKV Information Processing* 26
  - *Understanding Japanese Information Processing* xxi, 26
- O'Reilly, Tim xxiii
- Orwant, Jon xxiii
- Ousterhout, John 90, 189

P

- parentheses 12, 69, 83, 227-231
  - as `\(...\)` 57, 61
  - and alternation 12
  - balanced 126-127
  - capturing 218, 227, 290

    - and DFAs 93, 120
    - maintaining status 107, 150
    - only 95, 151
    - in Perl 36
  - capturing and efficiency 276, 282
  - counting 19, 44, 97, 229
  - disabling optimizations 156
  - efficiency 107, 150-153, 276, 282
  - grouping only 227
  - limiting scope 16
  - and `m/…/` 252-253
  - and `m/…/g` 237
  - matching balanced 126-127
  - mechanics of matching 93
  - and negative lookahead 229
  - nested 16, 40, 97
    - example 51
  - non-capturing 152, 227
  - and `split` 264
- **`ParseWords`** module 207, 278

Pascal 32, 54, 120
    format 35
    matching comments of 165
patch xxii
path
    isolating 132-136
`Path` module 278
Paxson, Vern xxii, 90
people
    Aho, Alfred 62, 90, 104, 183
    青山健治 xxiii
    Aoyama, Kenji xxiii
    Barwise, J. 60
    Beard, Paul xxiii
    Berke, Wayne xxii, 243
    Biggar, Mark xxii
    Brennan, Michael 90, 157, 185
    Chachich, Mike xxiii
    Christiansen, Tom xxii, 208
    Constable, Robert xxii, 60

Couch, Norris xxiii

Cruise, Tom 45

Dougherty, Dale 8

de Fermat, Pierre 86

Friedl

  Alfred 114

  brothers 30

  Jeffrey (home page) 309

  Liz 30

  parents xxi

Gosling, James xxii, 208

Haertel, Mike 90

Haight, Dick 90

Halpern, Jack xxi-xxiii, 3

春遍雀來 xxi-xxiii, 3

people (cont'd)

- Hietaniemi, Jarkko 242
- Joy, Bill 90
- Keisler, H. J. 60
- Kernighan, Brian xxii, 90, 183, 185, 311
- Kleene, Stephen 60
- Kleinedler, Steve xxii
- 小林劍 xxi-xxiii, 26
- Kunen, K. 60
- Lesk, Mike xxii, 90
- Lewine, Donald 243
- Libes, Don xxii, 89-90
- Lord, Tom 120
- Lunde, Ken xxi-xxiii, 26
- 増田清 xxiii
- Masuda, Keith xxiii
- Maton, William F. xxii, 32, 309
- McCulloch, Warren 60

McMahon, Lee 90

Mulder, Hans 258

Nerode, Anil xxii

Nudelman, Mark 90

Oram, Andy xxi-xxii, 5

O'Reilly, Tim xxiii

Orwant, Jon xxiii

Ousterhout, John 90, 189

Paxson, Vern xxii, 90

Pitts, Walter 60

Reilley, Chris xxiii

Ressler, Sandy 89

Robbins, Arnold 90, 184

Rochkind, Marc 183

van Rossum, Guido 90

Savarese, Daniel 311

Schienbrood, Eric 90

Schwartz, Randal 229, 258

Sethi, Ravi 104

- Spencer, Henry xxii-xxiii, 62, 120-121, 188, 208, 311
- Stallman, Richard xxii, 90, 198
- Steinbach, Paul 75
- Stok, Mike xxiii
- 高崎敬雄 xxiii
- Takasaki, Yukio xxiii
- Thompson, Ken xxii, 60-61, 78, 90
- Tubby 159-160, 165, 218, 285-287
- Ullman, Jeffrey 104
- Vromans, Johan 312

people (cont'd)

Wall, Larry xxii-xxiii, 33, 85, 90, 110, 203, 208-209, 225, 258, 304

- Weinberger, Peter 90, 183
- Welinder, Morten 312
- Wine, Hal 73
- Wood, Tom xxii

period (see dot)

Perl 90

- flavor overview 201
- `\1` 36, 219, 227, 243, 246, 272, 276, 279, 305

and substitution 218, 227, 243, 246, 284, 305

`\A` 236

`\B` 240-241, 297

`\b` 222, 240-241, 297

  dual meaning of 41, 76

`\E` 221, 240, 245

`\G` 217, 236-240, 292

  not the match start 251

`\l` 245

`\L` 245-246

`\Q` 221, 240, 245

`\s` 77, 241, 287

  introduction 42

  equivalence to character class 57

  and POSIX 242

`\U` 245-246

`\u` 245

`\Z` 236

`m/…/` (see Perl, match)

`s/…/…/` (see Perl, substitution)

`-DDEBUGGING` 155, 285

`<>` 49, 232, 234, 284, 289, 293

  and `$_` 55

`$+` 216-218, 257, 282-283

  and `!~` 251

  and `Benchmark` 284

  efficiency penalty 273-278, 281-282

  example 221, 290

  and lookahead 228

  and Perl4 305-307

  and `split` 263

  wishlist 304

`=~` 220, 248, 250

  introduction 34

  and `split` 262

Perl (cont'd)

- `$_` 55, 236, 250, 260, 272, 282
  - bug with study 289
- `?…?` 247, 251, 254-255, 308
- `$/` 31, 54, 232, 234, 293
- `$*` 234-235, 254
  - and quieting warnings 235
- `!~` 39, 251
- `$&`, `$'`, and `` $` `` 216-218, 256-257, 282-283
  - and `!~` 251
  - `` $` `` and `/g` 217
  - and `Benchmark` 284
  - `CheckNaughtiness` 279
  - efficiency penalty 273-278, 281-282
  - and lookahead 228
  - and Perl4 305, 307
  - and `split` 263
  - wishlist 83, 304

`$0` 218

`$1` 216-219, 257, 282-283

- introduction 36
- and `/g` 218, 247
- and `!~` 251
- and `Benchmark` 284
- efficiency penalty 273-278, 281-282
- and lookahead 228
- and Perl4 307
- and `split` 263
- wishlist 304

`?…?`

- in Perl4 306, 308

accidental feature 258

anchored 287

ANYOF 287

backreferences

- *vs.* octal escapes 243

character class 243-245, 279

  - and `\n` 235
  - and `/i` 280
  - and `/x` 223, 231, 295, 303
  - and backreferences 227, 243
  - instead of dot 226
  - *vs.* negative lookahead 228
  - shorthands 73, 241
- character shorthands 73, 241
- `CheckNaughtiness` 279
- comments within a regex 230-231
- context 210-211, 252-254
  - contorting 211
  - default for a match 251
  - list 210
  - list *vs.* scalar 256
  - responding to 211
- `cperl-mode` 194, 243
- dollar, variable interpolation 222
- dot instead of character class 226
- doublequoted string 241, 294

  - processing 221
  - variables appearing within 33, 219, 305
- doublequotish processing 219-224
  - and `\Q`, etc. 245
  - and `$` 232
  - avoiding redoing 268
  - bypassing 247
  - and replacement-operand 256
- efficiency 265-289
  - and parentheses 276, 282
  - penalty 273-282
  - substitution 281-284
- and Emacs 194, 243
- escape, unescapable 230
- `eval` 222, 235, 256, 279, 284, 308
  - and `/o` 270
  - string *vs.* block 271
- expression as a regex 248
- file slurp mode 234
- `foreach` *vs.* `while` *vs.* `if` 256

greatest weakness 203

`grep` 262

hexadecimal escapes 241, 243-244, 257

history 208, 258

home page 312

`if` *vs.* `while` *vs.* `foreach` 256

implicit 287

input record separator 31, 54, 232, 234, 293

interpolation 33, 219, 232, 246, 248, 255, 266, 283, 295, 304-305, 308

Just Another Perl Hacker 258

`lc` 44, 245

`lcfirst` 245

line anchor 82, 232-235, 287

Perl (cont'd)

- line mode
  - and `/m` 231-235, 249, 253
  - and `/s` 226, 231-235, 249
- literal text cognizance 286, 288
- `local` 212, 273-274
- `m/…/` (see Perl, match)
- maintenance nightmare 217, 228, 240, 306
- match 246-254
  - introduction 34
  - `?…?` 247, 251, 254-255, 308
  - `?…?` (Perl4) 306, 308
  - context 210-211, 251-254, 256
  - default context 251
  - default regex 248, 269
  - default regex (Perl4) 306
  - default regex (`split`) 264
  - general expression 248

  - identifying regex 222
  - match 36, 251, 263
  - operator 159
  - return value 252
  - side effects 36, 251, 263
  - and `split` 262
  - target operand 250
- maternal uncle of xxii
- `minlen` 155
- modifier 248
  - discussion 249-250
  - introduced 42
  - `/eieio` 258
  - `/osmosis` 249
  - specifying with `(?…)` 231
- modifier: cache compile `(/o)` 221, 224, 248, 266
  - discussion 268-273
  - and `eval` 270
  - penalty if forgotten 300

modifier: case insensitive `(/i)` 42, 231, 249

- and `$&` penalty 276
- and non-ASCII data 244
- benchmarked 280
- efficiency penalty 278-281
- and POSIX 242

modifier: evaluate replacement

- `(/e)` 255-256
- discussion 257-258
- and Perl4 bug 267
- and substitution efficiency 284

modifier: free-form `(/x)` 209, 221, 231, 249, 255, 297, 303

- and building regex in string 295
- and comments 231
- and delimiters 247
- how processed 223

modifier: global match `(/g)` 206, 252-254, 291

- and `\G` 236-240
- and `/i` penalty 280
- and `` $` `` 217

  - and `$&` penalty 276
  - and `$1` 218, 247
  - and context 252-254
  - and empty match 249-250, 259
  - and `pos` 239, 251
  - and `pos` (Perl4) 307
  - and `split` 260, 263
  - and substitution efficiency 283
  - as opposed to `while` 291
- modifier: multi-line mode `(/m)` 231-235, 249, 253
  - combining with `/s` (clean multi-line mode) 234-235
- modifier: single-line mode `(/s)` 226, 231-235, 249
- module
  - many naughty modules 278
  - `Carp` module 277-278
  - `Collate` module 242, 245
  - `English` module 277-278
  - `I18N::Collate` module 242, 245
  - `ParseWords` module 207, 278
  - `POXIS` module 242, 278

  `Text:: ParseWords` module 207, 278

motto 201, 266-267

multi-match anchor `(\G)` 217, 236-240, 292

  not the match start 251

`must have` 286, 288

`my` 211-212, 273-274

Perl (cont'd)

- non-greedy
  - quantifier 83, 291
- non-greedy quantifier 83, 206
- obfuscated 248, 258, 306
- octal escapes 222, 241, 243, 295
  - *vs.* backreferences 243
- optimization
  - `\G` 238
  - compile caching 224, 268-273
  - first-character discrimination 286
  - fixed-string check 280, 286, 288
  - implicit line anchor 287
- option
  - `-0` 32
  - `-c` 285
  - `-D` 155, 285
  - `-e` 32, 47, 285

- `-n` 32
- `-p` 47
- `-w` 34, 213, 285

parentheses 227-231

- capturing 227, 290
- capturing and efficiency 276, 282
- grouping only 227
- and `m/…/` 252-253
- and `m/…/g` 237
- and negative lookahead 229
- non-capturing 227
- and `split` 264

Perl

- option 155, 285

The Perl Way 201, 214, 265, 277

Perl4 *vs.* Perl5 208, 218, 225, 234-235, 239, 245, 249-250, 255, 257, 264, 267, 276, 285, 294, 305-308

`plus` 287

`pos` 209, 239, 251, 254

- Perl4 307

and POSIX locale 66, 209, 240, 242

- `\w` and `\s` 242

quantifier 225-227

- non-greedy 83, 291

`quotemeta` 245

rare characters 287

regex

- compile view of 285
- default 248, 269
- default (Perl4) 306
- default (split) 264
- delimiter 34, 43, 55, 220, 222-223, 230-231, 247-248, 255-256, 263, 296, 306-308
- using `$1` 219

replacement operand

- doublequoted string 255
- doublequoted string (Perl4) 307

return value

- match 252
- substitution 258

`s/…/…/` (see Perl, substitution)

`sawampersand` 279

scope

  dynamic 211-217

  lexical *vs.* dynamic 217

  limiting with parentheses 16

second-class metacharacters 246

shorthands 73, 241

side effects 36, 251, 263

  match 36, 251, 263

  and split 263

singlequotish processing 223, 247, 257, 307-308

`split` 204, 228, 259-266

  and `'●'` 263

  and capturing parentheses 264

  and match side effects 263

`start` 286, 288

`stclass` 287

strings as regexes 219-224

`study` 155, 254, 280, 287-289

    - bug 289
    - and the transmission 288
- substitution 255-259
    - introduction 45
    - and `\1` 218, 227, 243, 246, 284, 305
    - and `/e` 257
    - and `/g` 219
    - and `/i` 278
    - and `$1` 218
    - delimiter 255-256
    - delimiter (Perl4) 306-307
    - efficiency 281-284
    - and empty matches 250
    - and empty regex 259
    - identifying regex 222

Perl, substitution (cont'd)

    other methods 291

    replacement operand 255

    return value 258

    return value (Perl4) 308

  tied variables 217

  transmission

    and study 288

  `uc` 245

  `ucfirst` 245

  variable

    global *vs.* private 211-216

    interpolation 33, 219, 232, 246, 248, 255, 266, 283, 295, 304-305, 308

  version 5.000 problems 249, 257, 289

  version 5.001 problems 289

  warnings 34

    `-w` 34, 213, 285

    (`$^W` variable) 213, 235

        - temporarily turning off 213, 235
    - `while` *vs.* `foreach` *vs.* `if` 256
    - word anchor 45, 240-241, 292
- *Perl 5 Desktop Reference* 199, 312
- *The Perl Journal* 203, 226, 229, 312
- **`perl5db.pl`** package 278
- Pitts, Walter 60
- plus 17
    - as \+ 83
    - non-greedy 83
- **`Pod::Text`** module 278
- Portable Operating System Interface (see POSIX)
- **`pos`** 209, 239, 251, 254
    - Perl4 307
- POSIX
    - overview 64-66
    - `[:…:]` 80
    - `[=…=]` 80
    - `[.….]` 80-81
    - Basic Regular Expressions 64

bracket expressions 79

C library support 65

character class 80

character class and locale 80

character equivalent 80

collating sequences 65, 81

dot 78

escapes are literal 75

Extended Regular Expressions 64

locale 80

  overview 65

  and C library 65

  case-insensitive matching 65

  and Perl 66, 209, 240, 242

  and `\w` 65

missing support 66, 182

NFA (see POSIX NFA)

null character 78

program standards 89

  - superficial flavor chart 64

`Posix` module 242, 278

POSIX NFA

  - backtracking example 147
  - and efficiency 118
  - match cognizance optimization 157
  - testing for 145, 160-162

POSIX Programmer's Guide 243

possessive quantifiers 111, 297, 305

prepending filename to line 56

Principles of Compiler Design 104

private variable (see Perl, variable)

private variable *vs.* global 211-216

process ID 257

program standards (see POSIX)

Programming Perl 202-203

promote 211

pseudo-backtrack 106

publication

  - *Bulletin of Math. Biophysics* 60

Communications of the ACM 60

Compilers—Principles, Techniques, and Tools 104

Embodiments of Mind 60

The Kleene Symposium 60

lex &yacc 8

Life with Unix xxii, 89

"A logical calculus of the ideas immanent in nervous activity" 60

Mastering Regular Expressions 1-342

  online information 309

日本語情報処理 26

Perl 5 Desktop Reference 199, 312

The Perl Journal 203, 226, 229, 312

POSIX Programmer's Guide 243

publication (cont'd)

Principles of Compiler Design 104

Programming Perl 202-203

Regular Expression Search Algorithm 60

"The Role of Finite Automata in the Development of Modern Computing Theory" 60

Understanding CJKV Information Processing 26

Understanding Japanese Information Processing xxi, 26

Python 57, 69, 90, 96, 160

`\b`, dual meaning of 76

backreferences 70, 77

case-insensitive match 70

chart of shorthands 73

`compile` 57, 69, 96, 160

escape is passed string processing 77

flavor, changing 69

flavor resembles Emacs 69

`format` 35

`group(1)` 58, 70, 83, 96

`gsub` 58

home page 312

line anchor 58, 82

named subexpressions 228, 305

optimizations 152

quiet failures 165

snippet

  doubled-word example 57

  embolden HTML 69

  example 57

  filename and path 133

  `Subject` 96

  `"[Tt]ubby"` 160

strings as regexes 75

symbolic group names 228, 305

`\v99` 77

Q

Qantas 11

qed 61

- quantifier 16-18, 83-84
- 「(…*)*」 166
  - as \? 83
  - as \+ 83
  - and backtracking 106
  - can *always* match 108
  - character class
    - negated 226
  - efficiency 143, 155-156, 226
  - greediness 94
    - and alternation 112-114
    - deference to an overall match 98, 111, 131, 170, 226, 230
    - first come, first served 96, 98
    - local *vs.* global 120
    - non-greedy efficiency 152, 156
    - regex-directed 106
    - too greedy 97
  - interval 18, 84
    - non-greedy 83

  ⌈`x{0,0}`⌋ 84
minimal matching 83
multiple levels 166
non-greedy 83, 110, 206, 225
  alternation pitfalls 113
  efficiency 226
  interval 83
  vs. negated character class 226
  in Perl 83, 291
  plus 83
  question mark 83
  star 83
and parentheses 16
in Perl 83, 225-227, 291
plus 17
  as `\+` 83
  non-greedy 83
possessive 111, 297, 305
question mark 16
  as `\?` 83

        - non-greedy 83
    - smallest preceding subexpression 26
    - star 17
        - can *always* match 108
        - non-greedy 83
- ⌈`x{0,0}`⌋ 84
- question mark 16
    - as `\?` 83
    - non-greedy 83
- questions
    - reading ⌈`^cat$`⌋, ⌈`^$`⌋, … 8
    - ⌈`q[^u]`⌋ matching 'Qantas; 'Iraq' 11
    - `sizeless <HR>` 18

questions (cont'd)

  slicing a 24-hour time 23

  ⌈[·TAB]*⌋ *vs.* ⌈(·*|TAB*)⌋ 40

  checking for `c` or `C` 43

  `$var =~ s/\bJeff\b/Jeff/i` 45

  ⌈[0-9]*⌋ and 'a·1234·num' 106

  ⌈tour|to|tournament⌋ vs. ⌈to(ur(nament)?)?⌋? 112

  slice and dice a date 114

  simple change with ⌈[^"\\]+⌋ 141

  positive lookahead *vs.* negative lookahead 228

  `while` *vs.* `foreach` *vs.* `if` 254

  `local` *vs.* `my` 273

  you should be asking 63, 181, 185

quintessence of patience (see Friedl, Liz)

quotemeta 245

quotewords routine 207

R

range
  in class (see character class)
  of matches (see quantifier, interval)
rare characters 287
RE (see regex)
red dragon 104
regex
  building in a string
    and alternation 303
    in Perl 295
  comments 230-231
  compile 118, 155
  counting parentheses 19, 44, 97, 229
  delimiter 7, 61, 66-69
    in *awk* 187
    in Perl 34, 43, 55, 220, 222-223, 230-231, 247-248, 255-256, 263, 296, 306-308
  engine 24
    hybrid 121
    implementation ease 120

        listing of programs and types 90

        testing type 145, 160-162

    vs. English 171

    flavor 21

        changing 69

        hidden differences 184

        Perl 225-246

regex, flavor (cont'd)

        Python resembling Emacs 69

        questions you should be asking 63, 181, 185

        superficial chart 29, 63-64, 182-184, 189, 194, 201, 203, 225, 232, 236, 241, 245

        variety 63, 181-183, 185

    frame of mind 6

    free-form (see Perl, modifier: free-form `(/x)`)

    greediness 94

        and alternation 112-114

        deference to an overall match 98, 111, 131, 170, 226, 230

        first come, first served 96, 98

        local *vs.* global 120

  - non-greedy efficiency 152, 156
  - regex-directed 106
  - too greedy 97
- longest-leftmost match 91, 115, 117, 119, 184, 192
  - and POSIX 116
  - shortest-leftmost 121, 157
- match
  - function *vs.* other 159
- named subexpressions 228, 305
- nomenclature 24
- as an object 69
- odyssey xxi
- overall concept (description) 4
- parentheses
  - counting 19, 44, 97, 229
- provided as strings 75
  - in *awk* 187
  - in Emacs 69, 75
  - in Perl 219-224
  - in Python 75

    - in Tcl 75, 189, 193
- Python
    - named subexpressions 228, 305
    - Python 228, 305
    - *symbolic group names* 228, 305
- regex-directed 99, 101, 108, 139
    - alternation 122
    - and backreferences 219
    - control benefits 99
    - efficiency 118

regex, regex-directed (cont'd)
        essence 102
        and greediness 106
    rule 1: earliest match wins. 91
    rule 2: some metacharacters are
        greedy 94, 112
    subexpression
        counting 19, 44, 97, 229
        defined 25
        named 228, 305
    symbolic group names 228, 305
    text-directed 99-100
        efficiency 118
        regex appearance 101, 106
regex-directed matching 99, 101, 108, 139
    alternation 122
    and backreferences 219
    control benefits 99

  - efficiency 118
  - essence 102
  - and greediness 106
- regex-oriented case-insensitive implementation 278
- **`regexp`** 135, 159, 189-190
- **`regexp-quote`** 193
- **`regsub`** 68, 189, 191
- regular expression (see regex) origin of term 60
- *Regular Expression Search Algorithm* 60
- regular sets 60
- Reilley, Chris xxiii
- removing C comments 173, 292-293
- removing whitespace 282, 290-291
- replace (see substitution)
- **`replace-match`** 196-197
- replacement operand
  - doublequoted string 255
  - doublequoted string (Perl4) 307
- **`re-search-forward`** 68-69
- Ressler, Sandy 89

return value

    match 252

    substitution 258

RFC 822 294

rn xxii

Robbins, Arnold 90, 184

Rochkind, Marc 183

    "The Role of Finite Automata in the Development of Modern Computing Theory" 60

van Rossum, Guido 90

rules (see regex)

rummy

        playing 30

`rx` 120, 311

S

`s/`...**`/`**...**`/`** (see Perl, substitution; Perl, substitution)

`Safe` module 278

Savarese, Daniel 311

saved states (see backtracking)

`sawampersand` 279

scalar context (see Perl, context)

`Scalar` module 278

schaflopf 30

Schienbrood, Eric 90

Schwartz, Randal 229, 258

SCO

  grep 182

  Unix 185

scope

  dynamic 211-217

  lexical *vs.* dynamic 217

  limiting with parentheses 16

`SDBM_File` module 278

search and replace (see substitution) *sed* 90

  line anchors 82

`SelectSaver` module 278

`SelfLoader` module 278

separated text (see matching, delimited text)

Sethi, Ravi 104

shell 7

`Shell` module 278

shortest-leftmost 121, 157

shortest-leftmost match (see longest-left- most match)

side effects 179

- intertwined 39
- in Perl 36, 251, 263
- using 135

simple repetition optimization 192, 197

- discussed 155

- single-line mode (see Perl, modifier: single-line mode `(/s)`)
- singlequotish processing 223, 247, 257, 307-308
- **`Socket`** module 278
- software
  - getting 309
- some required (see quantifier, plus)
- space (see whitespace)
- *special* (see unrolling the loop)
- Spencer, Henry xxii-xxiii, 62, 120-121, 188, 208, 311
- **`split`**
  - in *awk* 187-188
  - in Perl 204, 228, 259-266
    - and `'●'` 263
    - and capturing parentheses 264
    - and match side effects 263
- square brackets (see character class)
- ⌋ 65, 81
- Stallman, Richard xxii, 90, 198

standard formula for matching delimited text 129, 169

standards (see POSIX)

star 17

  can *always* match 108

  non-greedy 83

star and friends (see quantifier)

start

  of line 8, 81

`start` 286, 288

start, of word (see word anchor)

states (see backtracking)

`stclass` 287

Steinbach, Paul 75

stock pricing example 46, 110

Stok, Mike xxiii

string (also see line)

  anchor (see line anchor)

  doublequoted (see doublequoted string)

  as regex

    in *awk* 187

        - in Emacs 69, 75
        - in Perl 219-224
        - in Python 75
        - in Tcl 75, 189, 193

string (cont'd)
    - string anchor optimization 92, 119, 158, 232, 287
    - (see also line)
    - string-oriented case-insensitive implementation 278

`study` 155, 254, 280, 287-289
    - bug 289
    - and the transmission 288

`sub`
    - in *awk* 68, 187

subexpression
    - counting 19, 44, 97, 229
    - defined 25
    - named 228, 305

substitution (see `sub`; `gsub`; `regsub`; `replace-match`)
    - in Perl (see Perl, substitution)
    - to remove text 111

    removing a partial match 47

`SubstrHash` module 278

symbolic group names 228, 305

syntax class 78

    and `isearch-forward` 192

    summary chart 195

`Sys::Hostname` module 278

`syslog` module 278

T

tabs (see `\t` (tab))

高崎敬雄 xxiii

Takasaki, Yukio xxiii

Tcl 90, 188-192

    `\1` 191

    `\9` 190

    `--` 191

    `-all` 68, 173, 175, 191

    backslash substitution processing 189

    case-insensitive match 68, 190-191

chart of shorthands 73

flavor chart 189

`format` 35

hexadecimal escapes 190

home page 312

`-indices` 135, 191, 304

line anchors 82

`\n` uncertainty 189

`-nocase` 68, 190-191

Tcl (cont'd)

  non-interpolative quoting 176

  null character 190

  octal escapes 190

  optimization 192

    compile caching 160

  support for POSIX locale 66

  `regexp` 135, 159, 189-190

  `regsub` 68, 189, 191

  snippet

    filename and path 133-135

    removing C comments 173-176

    `Subject` 96

    `[Tt]ubby` 159

  strings as regexes 75, 189, 193

  `\x0x`*ddd*… 190

temperature conversion 33-43, 199, 281

`Term::Cap` module 278

`Test::Harness` module 278

`testlib` module 278

text (see literal text)

`Text` module 278

text-directed matching 99-100

    efficiency 118

    regex appearance 101, 106

`Text::ParseWords` module 207, 278

`Text::Wrap` module 278

theory of an NFA 104

thingamajiggy as a technical term 71

Thompson, Ken xxii, 60-61, 78, 90

`Tie::Hash` module 278

`Tie::Scalar` module 278

`Tie::SubstrHash` module 278

time of day 23

`Time::Local` module 278

`time-now` 197

toothpicks

    scattered 69, 193

tortilla 65, 81

TPJ (see Perl)

trailing context 120

transmission (also see backtracking)

  and pseudo-backtrack 106

  keeping in synch 173, 206, 236-237, 290

  and `study` 288

  (see also backtracking)

Tubby 159-160, 165, 218, 285-287

twists and turns of optimization 177

type, global variable (see Perl, variable)

type, private variable (see Perl, variable)

U

`uc` 245

`ucfirst` 245

Ullman, Jeffrey 104

Understanding CJKV Information Processing 26

UnderstandingJapanese Information Processing xxi, 26

Unicode encoding 26

unlimited matches (see quantifier, star) unmatching (see backtracking)

unrolling the loop 162-172, 175, 265, 301

  checkpoint 164-166

  ⌈*normal** (*special normal**)*⌋ 164-165, 301

URL

  fetcher 73

V

`\v99` 77

variable

  in doublequoted strings 33, 219-224, 232, 245, 247, 256, 268

  global *vs.* private 211-216

  interpolation 33, 219, 232, 246, 248, 255, 266, 283, 295, 304-305, 308

  names 22

`vars` module 278

vertical tab (see \v (vertical tab))

vi 90

Vietnamese text processing 26

Virtual Software Library 310

Vromans, Johan 199, 312

W

Wall, Larry xxii-xxiii, 33, 85, 90, 110, 203, 208-209, 225, 258, 304

warnings 34

  `-w` 34, 213, 285

  (`$^W` variable) 213, 235

  setting `$*` 235

  temporarily turning off 213, 235

`webget` 73

Weinberger, Peter 90, 183

Welinder, Morten 312

`while` *vs.* `foreach` *vs.* `if` 256

whitespace 1

  allowing flexible 40

  allowing optional 17

  and *awk* 188

  in an email address 294

  as a regex delimiter 306

  removing 282, 290-291

  and `split` 263, 308

Windows-NT 185

Wine,Hal 73

Wood, Tom xxii

word anchor 14

  as lookbehind 230

  mechanics of matching 93

  in Perl 45, 240-241, 292

  sample line with positions marked 14

World Wide Web

  common CGI utility 33

  HTML 1, 9, 17-18, 69, 109, 127, 129, 229, 264

`wrap` module 278

WWW (see World Wide Web)

Y

Yahoo! 310

Z

zero or more (see quantifier, star)

ZIP (Zone Improvement Plan) code example 236-240

About the Author

Jeffrey Friedl was raised in the countryside of Rootstown, Ohio, and had aspirations of being an astronomer until one day noticing a TRS-80 Model I sitting unused in the corner of the chem lab (bristling with a *full* 16k RAM, no less). He eventually began using UNIX (and regular expressions) in 1980. With degrees in computer science from Kent (B.S.) and the University of New Hampshire (M.S.), he is now an engineer with Omron Corporation, Kyoto, Japan. He lives in Nagaokakyou-city with Tubby, his long-time friend and Teddy Bear, in a tiny apartment designed for a (Japanese) family of three.

Jeffrey applies his regular-expression know-how to make the world a safer place for those not bilingual in English and Japanese. He built and maintains the World Wide Web Japanese-English dictionary server, *http.//www.itc.omron.com/cgi-bin/j-e*, and is active in a variety of language-related projects, both in print and on the Web.

When faced with the daunting task of filling his copious free time, Jeffrey enjoys riding through the mountainous countryside of Japan on his Honda CB-1. At the age of 30, he finally decided to put his 6'4" height to some use, and joined the Omron company basketball team. While finalizing the manuscript for *Mastering Regular Expressions,* he took time out to appear in his first game, scoring five points in nine minutes of play, which he feels is pretty darn good for a geek. When visiting his family in The States, Jeffrey enjoys dancing a two-step with his mom, binking old coins with his dad, and playing *schoffkopf* with his brothers and sisters.

Colophon

The birds featured on the cover of *Mastering Regular Expressions* are owls. There are two families and approximately 180 species of these birds of prey distributed throughout the world, with the exception of Antarctica. Most species of owl are nocturnal hunters, feeding entirely on live animals, ranging in size from insects to hares.

Because they have little ability to move their large, forward-facing eyes, owls must move their entire heads in order to look around. They can rotate their heads up to 270 degrees, and some can turn their heads completely upside down. Among the physical adaptations that enhance owls' effectiveness as hunters is their extreme sensitive to the frequency and direction of sounds. Many species of owl have asymmetrical ear placement, which enables them to more easily locate their prey in dim or dark light. Once they've pinpointed the location, the owl's soft feathers allow them to fly noiselessly and thus to surprise their prey.

While people have traditionally anthropomorphized birds of prey as evil and coldblooded creatures, owls are viewed differently in human mythology. Perhaps because their large eyes give them the appearance of intellectual depth, owls have been portrayed in folklore through the ages as wise creatures.

Edie Freedman designed this cover and the entire UNIX bestiary that appears on Nutshell Handbooks, using a 19th-century engraving from the Dover Pictorial Archive. The cover layout was produced with Quark XPress 3.3 using the ITC Garamond font.

The text was prepared by Jeffrey Friedl in a hybrid markup of his own design, mixing SGML, raw troff, raw PostScript, and his own markup. A home-grown filter translated the latter to the other, lower-level markups, the result of which was processed by a locally modified version of O'Reilly's SGML tools (this step requiring upwards of an hour of raw processing time, and over 75 megabytes of process space, just for Chapter 7!). That result was then processed by a locally-modified version of James Clark's *gtroff* producing camera-ready PostScript for O'Reilly.

The text was written and processed on an IBM ThinkPad 755 CX, provided by Omron Corporation, running Linux the X Windows System, and Mule (Multilingual Emacs). A notoriously poor speller, Jeffrey made heavy use of *ispell* and its Emacs interface. For imaging during development, Jeffrey used Ghostscript (from Aladdin Enterprises, Menlo Park, California), as well as an Apple Color Laser Writer 12/600PS provided by Omron. Test prints at 1270dpi were kindly provided by Ken Lunde, of Adobe Systems, using a Linotronic L300-J.

Ken Lunde also provided a number of special font and typesetting needs, including custom-designed characters and Japanese characters from Adobe Systems's Heisei Mincho W3 typeface.

The figures were originally created by Jeffrey using *xfig,* as well as Thomas Williams's and Colin Kelley's *gnuplot.* They were then greatly enhanced by Chris Reilley using Macromedia Freehand.

The text is set in ITC Garamond Light; code is set in ConstantWillison; figure labels are in Helvetica Black.