

Pro Perl Parsing



Christopher M. Frenz

Pro Perl Parsing

Copyright © 2005 by Christopher M. Frenz

Lead Editors: Jason Gilmore and Matthew Moodie

Technical Reviewer: Teodor Zlatanov

Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis,

Jason Gilmore, Jonathan Hassell, Chris Mills, Dominic Shakeshaft, Jim Sumser

Associate Publisher: Grace Wong

Project Manager: Beth Christmas

Copy Edit Manager: Nicole LeClerc

Copy Editor: Kim Wimpsett

Assistant Production Director: Kari Brooks-Copony

Production Editor: Laura Cheu

Compositor: Linda Weidemann, Wolf Creek Press

Proofreader: Nancy Sixsmith

Indexer: Tim Tate

Artist: Wordstop Technologies Pvt. Ltd., Chennai

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Library of Congress Cataloging-in-Publication Data

Frenz, Christopher.

Pro Perl parsing / Christopher M. Frenz.

p. cm.

Includes index.

ISBN 1-59059-504-1 (hardcover : alk. paper)

1. Perl (Computer program language) 2. Natural language processing (Computer science) I. Title.

QA76.73.P22F72 2005

005.13'3--dc22

2005017530

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit <http://www.springeronline.com>.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit <http://www.apress.com>.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at <http://www.apress.com> in the Downloads section.

For Jonathan!
You are the greatest son
any father could ask for.

Contents at a Glance

About the Author	xiii
About the Technical Reviewer	xv
Acknowledgments	xvii
Introduction	xix
■ CHAPTER 1 Parsing and Regular Expression Basics	1
■ CHAPTER 2 Grammars	37
■ CHAPTER 3 Parsing Basics	63
■ CHAPTER 4 Using Parse::Yapp	85
■ CHAPTER 5 Performing Recursive-Descent Parsing with Parse::RecDescent	109
■ CHAPTER 6 Accessing Web Data with HTML::TreeBuilder	137
■ CHAPTER 7 Parsing XML Documents with XML::LibXML and XML::SAX	161
■ CHAPTER 8 Introducing Miscellaneous Parsing Modules	185
■ CHAPTER 9 Finding Solutions to Miscellaneous Parsing Problems	201
■ CHAPTER 10 Performing Text and Data Mining	217
■ INDEX	243

Contents

- About the Author xiii
- About the Technical Reviewer xv
- Acknowledgments xvii
- Introduction xix

- CHAPTER 1 Parsing and Regular Expression Basics 1**
 - Parsing and Lexing 2
 - Parse::Lex 4
 - Using Regular Expressions 6
 - A State Machine 7
 - Pattern Matching 12
 - Quantifiers 14
 - Predefined Subpatterns 15
 - Posix Character Classes 16
 - Modifiers 17
 - Assertions 20
 - Capturing Substrings 24
 - Substitution 26
 - Troubleshooting Regexes 26
 - GraphViz::Regex 27
 - Using Regexp::Common 28
 - Regexp::Common::Balanced 29
 - Regexp::Common::Comments 30
 - Regexp::Common::Delimited 30
 - Regexp::Common::List 30
 - Regexp::Common::Net 31
 - Regexp::Common::Number 31
 - Universal Flags 32
 - Standard Usage 32
 - Subroutine-Based Usage 33
 - In-Line Matching and Substitution 34
 - Creating Your Own Expressions 35
 - Summary 36

CHAPTER 2	Grammars	37
	Introducing Generative Grammars	38
	Grammar Recipes	39
	Sentence Construction	41
	Introducing the Chomsky Method	42
	Type 1 Grammars (Context-Sensitive Grammars)	44
	Type 2 Grammars (Context-Free Grammars)	48
	Type 3 Grammars (Regular Grammars)	54
	Using Perl to Generate Sentences	55
	Perl-Based Sentence Generation	56
	Avoiding Common Grammar Errors	59
	Generation vs. Parsing	60
	Summary	61
CHAPTER 3	Parsing Basics	63
	Exploring Common Parser Characteristics	64
	Introducing Bottom-Up Parsers	65
	Coding a Bottom-Up Parser in Perl	68
	Introducing Top-Down Parsers	73
	Coding a Top-Down Parser in Perl	74
	Using Parser Applications	78
	Programming a Math Parser	80
	Summary	83
CHAPTER 4	Using Parse::Yapp	85
	Creating the Grammar File	85
	The Header Section	86
	The Rule Section	87
	The Footer Section	88
	Using yapp	94
	The -v Flag	99
	The -m Flag	103
	The -s Flag	103
	Using the Generated Parser Module	104
	Evaluating Dynamic Content	105
	Summary	108

CHAPTER 5	Performing Recursive-Descent Parsing with Parse::RecDescent	109
	Examining the Module's Basic Functionality	109
	Constructing Rules	111
	Subrules	112
	Introducing Actions	115
	@item and %item	116
	@arg and %arg	117
	\$return	118
	\$text	120
	\$thisline and \$prevline	120
	\$thiscolumn and \$prevcolumn	121
	\$thisoffset and \$prevoffset	121
	\$thisparser	121
	\$thisrule and \$thisprod	122
	\$score	122
	Introducing Startup Actions	122
	Introducing Autoactions	124
	Introducing Autotrees	125
	Introducing Autostubbing	127
	Introducing Directives	128
	<commit> and <uncommit>	129
	<reject>	130
	<skip>	131
	<resync>	132
	<error>	132
	<defer>	132
	<perl>	133
	<score> and <autoscore>	134
	Precompiling the Parser	135
	Summary	135

CHAPTER 6	Accessing Web Data with HTML::TreeBuilder	137
	Introducing HTML Basics	137
	Specifying Titles	138
	Specifying Headings	139
	Specifying Paragraphs	140
	Specifying Lists	141
	Embedding Links	142
	Understanding the Nested Nature of HTML	143
	Accessing Web Content with LWP	145
	Using LWP::Simple	146
	Using LWP	146
	Using HTML::TreeBuilder	150
	Controlling TreeBuilder Parser Attributes	152
	Searching Through the Parse Tree	154
	Understanding the Fair Use of Information Extraction Scripts	158
	Summary	159
 CHAPTER 7	 Parsing XML Documents with XML::LibXML and XML::SAX	 161
	Understanding the Nature and Structure of XML Documents	163
	The Document Prolog	164
	Elements and the Document Body	166
	Introducing Web Services	172
	XML-RPC	173
	RPC::XML	173
	Simple Object Access Protocol (SOAP)	174
	SOAP::Lite	175
	Parsing with XML::LibXML	177
	Using DOM to Parse XML	177
	Parsing with XML::SAX::ParserFactory	179
	Summary	182
 CHAPTER 8	 Introducing Miscellaneous Parsing Modules	 185
	Using Text::Balanced	185
	Using extract_delimited	186
	Using extract_bracketed	188
	Using extract_codeblock	189

Using extract_quotelike	190
Using extract_variable	191
Using extract_multiple	192
Using Date::Parse	193
Using XML::RSS::Parser	194
Using Math::Expression	197
Summary	199
CHAPTER 9 Finding Solutions to Miscellaneous Parsing Problems	201
Parsing Command-Line Arguments	201
Parsing Configuration Files	204
Refining Searches	205
Formatting Output	212
Summary	214
CHAPTER 10 Performing Text and Data Mining	217
Introducing Data Mining Basics	218
Introducing Descriptive Modeling	219
Clustering	219
Summarization	220
Association Rules	221
Sequence Discovery	224
Introducing Predictive Modeling	224
Classification	225
Regression	225
Time Series Analysis	228
Prediction	229
Summary	241
INDEX	243

About the Author

■ **CHRISTOPHER M. FRENZ** is currently a bioinformaticist at New York Medical College. His research interests include applying artificial neural networks to protein engineering as well using molecular modeling techniques to determine the role that protein structures have on protein function. Frenz uses the Perl programming language to conduct much of his research. Additionally, he is the author of *Visual Basic and Visual Basic .NET for Scientists and Engineers* (Apress, 2002) as well as numerous scientific and computer articles. Frenz has more than ten years of programming experience and, in addition to Perl and VB, is also proficient in the Fortran and C++ languages. Frenz can be contacted at cfrenz@gmail.com.

About the Technical Reviewer

■ **TEODOR ZLATANOV** earned his master's degree in computer engineering from Boston University in 1999 and has been happily hacking ever since. He always wonders how it is possible to get paid for something as fun as programming, but tries not to make too much noise about it.

Zlatanov lives with his wife, 15-month-old daughter, and two dogs, Thor and Maple, in lovely Braintree, Massachusetts. He wants to thank his family for their support and for the inspiration they always provide.

Acknowledgments

Bringing this book from a set of ideas to the finished product that you see before you today would not have been possible without the help of others. Jason Gilmore was a great source of ideas for refining the content of the early chapters in this book, and Matthew Moodie provided equally insightful commentary for the later chapters and assisted in ensuring that the final page layouts of the book looked just right. I am also appreciative of Teodor Zlatanov's work as a technical reviewer, since he went beyond the role of simply finding technical inaccuracies and made many valuable suggestions that helped improve the clarity of the points made in the book. Beth Christmas also played a key role as the project manager for the entire process; without her friendly prompting, this book would probably still be in draft form. I would also like to express my appreciation of the work done by Kim Wimpsett and Laura Cheu, who did an excellent job preparing the manuscript and the page layouts, respectively, for publication. Last, but not least, I would like to thank my family for their support on this project, especially my wife, Thao, and son, Jonathan.

Introduction

Over the course of the past decade, we have all been witnesses to an explosion of information, in terms of both the amounts of knowledge that exists within the world and the availability of such information, with the proliferation of the World Wide Web being a prime example. Although these advancements of knowledge have undoubtedly been beneficial, they have also created new challenges in information retrieval, in information processing, and in the extraction of relevant information. This is in part due to a diversity of file formats as well as the proliferation of loosely structured formats, such as HTML. The solution to such information retrieval and extraction problems has been to develop specialized parsers to conduct these tasks. This book will address these tasks, starting with the most basic principles of data parsing.

The book will begin with an introduction to parsing basics using Perl's regular expression engine. Once these regex basics are mastered, the book will introduce the concept of generative grammars and the Chomsky hierarchy of grammars. Such grammars form the base set of rules that parsers will use to try to successfully parse content of interest, such as text or XML files. Once grammars are covered, the book proceeds to explain the two basic types of parsers—those that use a top-down approach and those that use a bottom-up approach to parsing. Coverage of these parser types is designed to facilitate the understanding of more powerful parsing modules such as Yapp (bottom-up) and RecDescent (top-down).

Once these powerful and flexible generalized parsing modules are covered, the book begins to delve into more specialized parsing modules such as parsing modules designed to work with HTML. Within Chapter 6, the book also provides an overview of the LWP modules, which facilitate access to documents posted on the Web. The parsing examples within this chapter will use the LWP modules to parse data that is directly accessed from the Web. Next the book examines the parsing of XML data, which is a markup language that is increasingly growing in popularity. The XML coverage also discusses SOAP and XML-RPC, which are two of the most popular methods for accessing remote XML-formatted data. The book then covers several smaller parsing modules, such as an RSS parser and a date/time parser, as well as some useful parsing tasks, such as the parsing of configuration files. Lastly, the book introduces data mining. *Data mining* provides a means for individuals to work with extracted data (as well as other types of data) so that the data can be used to learn more about a given area or to make predictions about future directions that area of interest may take. This content aims to demonstrate that although parsing is often a critical data extraction and retrieval task, it may just be a component of a larger data mining system.

This book examines all these problems from the perspective of the Perl programming language, which, since its inception in 1987, has always been heralded for its parsing and text processing capabilities. The book takes a practical approach to parsing and is rich in examples that are relevant to real-world parsing tasks. While covering all the basics of parser design to instill understanding in readers, the book highlights numerous CPAN modules that will allow programmers to produce working parser code in an efficient manner.



Parsing and Regular Expression Basics

The dawn of a new age is upon us, an information age, in which an ever-increasing and seemingly endless stream of new information is continuously generated. Information discovery and knowledge advancements occur at such rates that an ever-growing number of specialties is appearing, and in many fields it is impossible even for experts to master everything there is to know. Anyone who has ever typed a query into an Internet search engine has been a firsthand witness to this information explosion. Even the most mundane terms will likely return hundreds, if not thousands, of hits. The sciences, especially in the areas of genomics and proteomics, are generating seemingly insurmountable mounds of data.

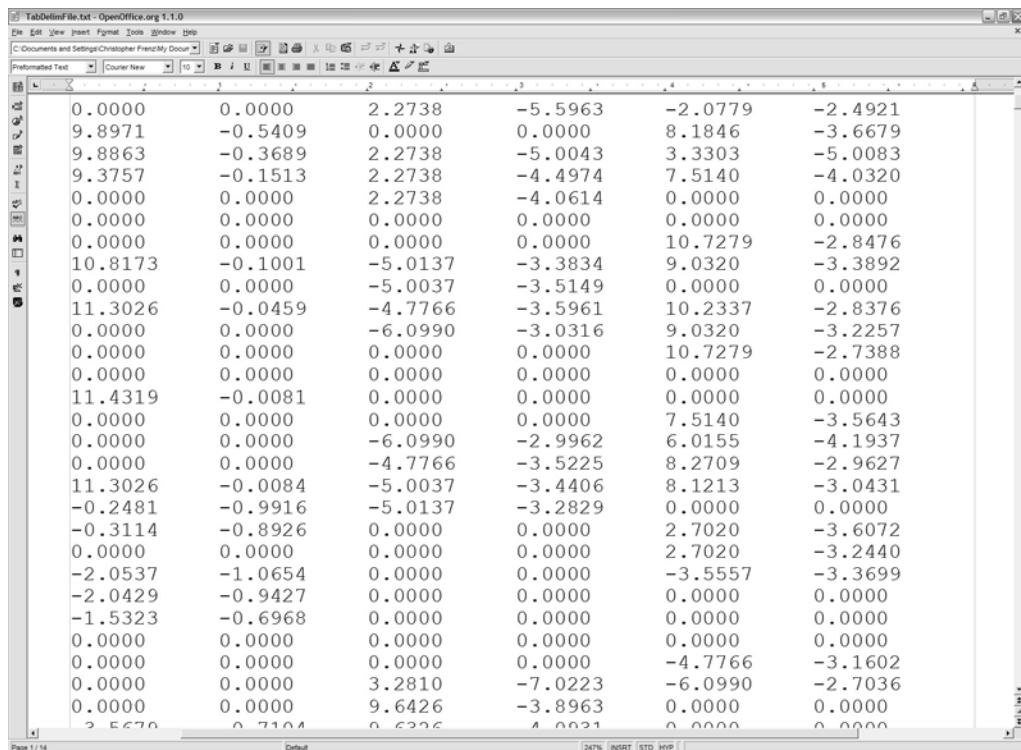
Yet, one must also consider that this generated data, while not easily accessible to all, is often put to use, resulting in the creation of new ideas to generate even more knowledge or in the creation of more efficient means of data generation. Although the old adage “knowledge is power” holds true, and almost no one will deny that the knowledge gained has been beneficial, the sheer volume of information has created quite a quandary. Finding information that is exactly relevant to your specific needs is often not a simple task. Take a minute to think about how many searches you performed in which all the hits returned were both useful and easily accessible (for example, were among the top matches, were valid links, and so on). More than likely, your search attempts did not run this smoothly, and you needed to either modify your query or buckle down and begin to dig for the resources of interest.

Thus, one of the pressing questions of our time has been how do we deal with all of this data so we can efficiently find the information that is currently of interest to us? The most obvious answer to this question has been to use the power of computers to store these giant catalogs of information (for example, databases) and to facilitate searches through this data. This line of reasoning has led to the birth of various fields of informatics (for example, bioinformatics, health informatics, business informatics, and so on). These fields are geared around the purpose of developing powerful methods for storing and retrieving data as well as analyzing it.

In this book, I will explain one of the most fundamental techniques required to perform this type of data extraction and analysis, the technique of *parsing*. To do this, I will show how to utilize the Perl programming language, which has a rich history as a powerful text processing language. Furthermore, Perl is already widely used in many fields of informatics, and many robust parsing tools are readily available for Perl programmers in the form of CPAN modules. In addition to examining the actual parsing methods themselves, I will also cover many of these modules.

Parsing and Lexing

Before I begin covering how you can use Perl to accomplish your parsing tasks, it is essential to have a clear understanding of exactly what parsing is and how you can utilize it. Therefore, I will define *parsing* as the action of splitting up a data set into smaller, more meaningful units and uncovering some form of meaningful structure from the sequence of these units. To understand this point, consider the structure of a tab-delimited data file. In this type of file, data is stored in columns, and a tab separates consecutive columns (see Figure 1-1).



0.0000	0.0000	2.2738	-5.5963	-2.0779	-2.4921
9.8971	-0.5409	0.0000	0.0000	8.1846	-3.6679
9.8863	-0.3689	2.2738	-5.0043	3.3303	-5.0083
9.3757	-0.1513	2.2738	-4.4974	7.5140	-4.0320
0.0000	0.0000	2.2738	-4.0614	0.0000	0.0000
0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
0.0000	0.0000	0.0000	0.0000	10.7279	-2.8476
10.8173	-0.1001	-5.0137	-3.3834	9.0320	-3.3892
0.0000	0.0000	-5.0037	-3.5149	0.0000	0.0000
11.3026	-0.0459	-4.7766	-3.5961	10.2337	-2.8376
0.0000	0.0000	-6.0990	-3.0316	9.0320	-3.2257
0.0000	0.0000	0.0000	0.0000	10.7279	-2.7388
0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
11.4319	-0.0081	0.0000	0.0000	0.0000	0.0000
0.0000	0.0000	0.0000	0.0000	7.5140	-3.5643
0.0000	0.0000	-6.0990	-2.9962	6.0155	-4.1937
0.0000	0.0000	-4.7766	-3.5225	8.2709	-2.9627
11.3026	-0.0084	-5.0037	-3.4406	8.1213	-3.0431
-0.2481	-0.9916	-5.0137	-3.2829	0.0000	0.0000
-0.3114	-0.8926	0.0000	0.0000	2.7020	-3.6072
0.0000	0.0000	0.0000	0.0000	2.7020	-3.2440
-2.0537	-1.0654	0.0000	0.0000	-3.5557	-3.3699
-2.0429	-0.9427	0.0000	0.0000	0.0000	0.0000
-1.5323	-0.6968	0.0000	0.0000	0.0000	0.0000
0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
0.0000	0.0000	3.2810	-7.0223	-6.0990	-2.7036
0.0000	0.0000	9.6426	-3.8963	0.0000	0.0000
2.5670	0.7104	0.6226	4.0021	0.0000	0.0000

Figure 1-1. A tab-delimited file

Reviewing this file, your eyes most likely focus on the numbers in each column and ignore the whitespace found between the columns. In other words, your eyes perform a parsing task by allowing you to visualize distinct columns of data. Rather than just taking the whole data set as a unit, you are able to break up the data set into columns of numbers that are much more meaningful than a giant string of numbers and tabs. While this example is simplistic, we carry out parsing actions such as this every day. Whenever we see, read, or hear anything, our brains must parse the input in order to make some kind of logical sense out of it. This is why parsing is such a crucial technique for a computer programmer—there will often be a need to parse data sets and other forms of input so that applications can work with the information presented to them.

The following are common types of parsed data:

- Data TypeText files
- CSV files
- HTML
- XML
- RSS files
- Command-line arguments
- E-mail/Web page headers
- HTTP headers
- POP3 headers
- SMTP headers
- IMAP headers

To get a better idea of just how parsing works, you first need to consider that in order to parse data you must classify the data you are examining into units. These units are referred to as *tokens*, and their identification is called *lexing*. In Figure 1-1, the units are numbers, and a tab separates each unit; for many lexing tasks, such whitespace identification is adequate. However, for certain sophisticated parsing tasks, this breakdown may not be as straightforward. A recursive approach may also be warranted, since in more nested structures it becomes possible to find units within units. Math equations such as $4*(3+2)$ provide an ideal example of this. Within the parentheses, 3 and 2 behave as their own distinct units; however, when it comes time to multiply by 4, (3+2) can be considered as a single unit. In fact, it is in dealing with nested structures such as this example

that full-scale parsers prove their worth. As you will see later in the “Using Regular Expressions” section, simpler parsing tasks (in other words, those with a known finite structure) often do not require full-scale parsers but can be accomplished with regular expressions and other like techniques.

Note Examples of a well-known lexer and parser are the C-based Lex and Yacc programs that generally come bundled with Unix-based operating systems.

Parse::Lex

Before moving on to more in-depth discussions of parsers, I will introduce the Perl module `Parse::Lex`, which you can use to perform lexing tasks such as lexing the math equation listed previously.

Tip `Parse::Lex` and the other Perl modules used in this book are all available from CPAN (<http://www.cpan.org>). If you are unfamiliar with working with CPAN modules, you can find information about downloading and installing Perl modules on a diversity of operating systems at <http://search.cpan.org/~jhi/perl-5.8.0/pod/perlmodinstall.pod>. If you are using an ActiveState Perl distribution, you can also install Perl modules using the Perl Package Manager (PPM). You can obtain information about its use at <http://aspn.activestate.com/ASPN/docs/ActivePerl/faq/ActivePerl-faq2.html>.

For more detailed information about CPAN and about creating and using Perl modules, you will find that *Writing Perl Modules for CPAN* (Apress, 2002) by Sam Tregar is a great reference.

Philippe Verdret authored this module; the most current version as of this book's publication is version 2.15. `Parse::Lex` is an object-oriented lexing module that allows you to split input into various tokens that you define. Take a look at the basics of how this module works by examining Listing 1-1, which will parse simple math equations, such as `18.2+43/6.8`.

Listing 1-1. Using `Parse::Lex`

```
#!/usr/bin/perl

use Parse::Lex;
```

```
#defines the tokens
@token=qw(
    BegParen  [\()]
    EndParen  [\)]
    Operator  [-+*/^]
    Number    [-?\d+|-?\d+\.\d*]
);
$lexer=Parse::Lex->new(@token); #Specifies the lexer
$lexer->from(STDIN); #Specifies the input source

TOKEN:
while(1){ #1 will be returned unless EOI
    $token=$lexer->next;
    if(not $lexer->eoi){
        print $token->name . " " . $token->text . " " . "\n";
    }
    else {last TOKEN;}
}
```

The first step in using this module is to create definitions of what constitutes an acceptable token. Token arguments for this module usually consist of a token name argument, such as the previous `BegParen`, followed by a regular expression. Within the module itself, these tokens are stored as instances of the `Parse::Token` class. After you specify your tokens, you next need to specify how your lexer will operate. You can accomplish this by passing a list of arguments to the lexer via the `new` method. In Listing 1-1, this list of arguments is contained in the `@token` array. When creating the argument list, it is important to consider the order in which the token definitions are placed, since an input value will be classified as a token of the type that it is first able to match. Thus, when using this module, it is good practice to list the strictest definitions first and then move on to the more general definitions. Otherwise, the general definitions may match values before the stricter comparisons even get a chance to be made.

Once you have specified the criteria that your lexer will operate on, you next define the source of input into the lexer by using the `from` method. The default for this property is `STDIN`, but it could also be a filename, a file handle, or a string of text (in quotes). Next you loop through the values in your input until you reach the `eoi` (end of input) condition and print the token and corresponding type. If, for example, you entered the command-line argument `43.4*15^2`, the output should look like this:

```
Number 43.4
Operator *
Number 15
Operator ^
Number 2
```

In Chapter 3, where you will closely examine the workings of full-fledged parsers, I will employ a variant of this routine to aid in building a math equation parser.

Regular expressions are one of the most useful tools for lexing, but they are not the only method. As mentioned earlier, for some cases you can use whitespace identification, and for others you can bring dictionary lists into play. The choice of lexing method depends on the application. For applications where all tokens are of a similar type, like the tab-delimited text file discussed previously, whitespace pattern matching is probably the best bet. For cases where multiple token types may be employed, regular expressions or dictionary lists are better bets. For most cases, regular expressions are the best since they are the most versatile. Dictionary lists are better suited to more specialized types of lexing, where it is important to identify only select tokens.

One such example where a dictionary list is useful is in regard to the recent bioinformatics trend of mining medical literature for chemical interactions. For instance, many scientists are interested in the following:

```
<Chemical A> <operates on> <Chemical B>
```

In other words, they just want to determine how chemical A interacts with chemical B. When considering this, it becomes obvious that the entire textual content of any one scientific paper is not necessary to tokenize and parse. Thus, an informatician coding such a routine might want to use dictionary lists to identify the chemicals as well as to identify terms that describe the interaction. A dictionary list would be a listing of all the possible values for a given element of a sentence. For example, rather than `operates on`, I could also fill in `reacts with`, `interacts with`, or a variety of other terms and have a program check for the occurrence of any of those terms. Later, in the section “Capturing Substrings,” I will cover this example in more depth.

Using Regular Expressions

As you saw in the previous `Parse::Lex` example, regular expressions provide a robust tool for token identification, but their usefulness goes far beyond that. In fact, for many simple parsing tasks, a regular expression alone may be adequate to get the job done. For example, if you want to perform a simple parsing/data extraction task such as parsing out an e-mail address found on a Web page, you can easily accomplish this by using a regular expression. All you need is to create a regular expression that identifies a pattern similar to the following:

```
[alphanumeric characters]@[alphanumeric characters.com]
```

Caution The previous expression is a simplification provided to illustrate the types of pattern matching for which you can use regular expressions. A more real-world e-mail matching expression would need to be more complex to account for other factors such as alternate endings (for example, .net, .gov) as well as the presence of metacharacters in either alphanumeric string. Additionally, a variety of less-common alternative e-mail address formats may also warrant consideration.

The following sections will explain how to create such regular expressions in the format Perl is able to interpret. To make regular expressions and their operation a little less mysterious, however, I will approach this topic by first explaining how Perl's regular expression engine operates. Perl's regular expression engine functions by using a programming paradigm known as a *state machine*, described in depth next.

A State Machine

A simple definition of a state machine is one that will sequentially read in the symbols of an input word. After reading in a symbol, it will decide whether the current state of the machine is one of acceptance or nonacceptance. The machine will then read in the next symbol and make another state decision based upon the previous state and the current symbol. This process will continue until all symbols in the word are considered. Perl's regular expression engine operates as a state machine (sometimes referred to as an *automaton*) for a given string sequence (that is, the word). In order to match the expression, all of the acceptable states (that is, characters defined in the regular expression) in a given path must be determined to be true. Thus, when you write a regular expression, you are really providing the criteria the differing states of the automaton need to match in order to find a matching string sequence. To clarify this, let's consider the pattern /123/ and the string 123 and manually walk through the procedure the regular expression engine would perform. Such a pattern is representative of the simplest type of case for your state machine. That is, the state machine will operate in a completely linear manner. Figure 1-2 shows a graphical representation of this state machine.

Note It is interesting to note that a recursive descent parser evaluates the regular expressions you author. For more information on recursive descent parsers, see Chapter 5.

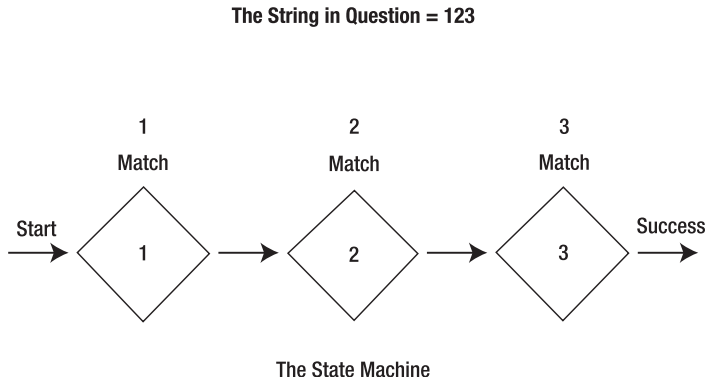


Figure 1-2. A state machine designed to match the pattern `/123/`

In this case, the regular expression engine begins by examining the first character of the string, which is a 1. In this case, the required first state of the automaton is also a 1. Therefore, a match is found, and the engine moves on by comparing the second character, which is a 2, to the second state. Also in this case, a match is found, so the third character is examined and another match is made. When this third match is made, all states in the state machine are satisfied, and the string is deemed a match to the pattern.

In this simple case, the string, as written, provided an exact match to the pattern. Yet, this is hardly typical in the real world, so it is important to also consider how the regular expression will operate when the character in question does not match the criterion of a particular state in the state machine. In this instance, I will use the same pattern (`/123/`) and hence the same state machine as in the previous example, only this time I will try to find a match within the string 4512123 (see Figure 1-3).

This time the regular expression engine begins by comparing the first character in the string, 4, with the first state criterion. Since the criterion is a 1, no match is found. When this mismatch occurs, the regular expression starts over by trying to compare the string contents beginning with the character in the second position (see Figure 1-4).

As in the first case, no match is found between criterion for the first state and the character in question (5), so the engine moves on to make a comparison beginning with the third character in the string (see Figure 1-5).

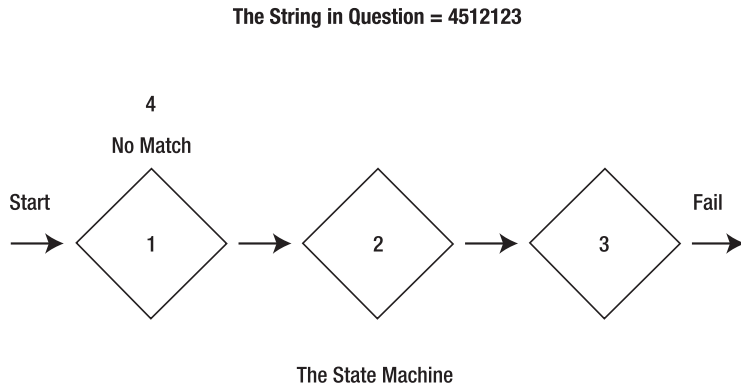


Figure 1-3. *The initial attempt at comparing the string 4512123 to the pattern /123/*

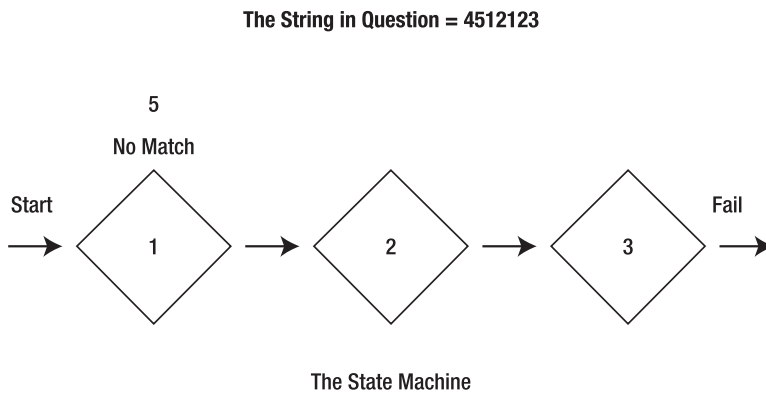


Figure 1-4. *The second attempt at comparing the string 4512123 to the pattern /123/*

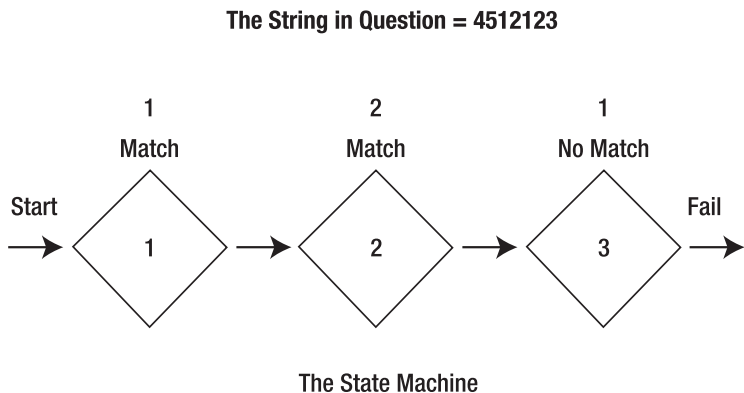


Figure 1-5. *The third attempt at comparing the string 4512123 to the pattern /123/*

In this case, since the third character is a 1, the criterion for the first state is satisfied, and thus the engine is able to move on to the second state. The criterion for the second state is also satisfied, so therefore the engine will next move on to the third state. The 1 in the string, however, does not match the criterion for state 3, so the engine then tries to match the fourth character of the string, 2, to the first state (see Figure 1-6).

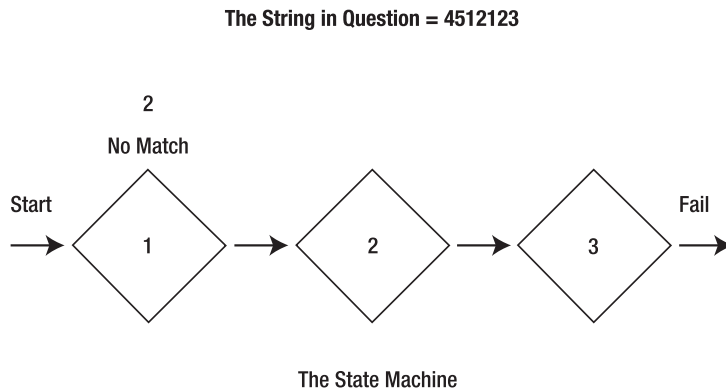


Figure 1-6. The fourth attempt at comparing the string 4512123 to the pattern /123/

As in previous cases, the first criterion is not satisfied by the 2, and consequently the regular expression engine will begin to examine the string beginning with the fifth character. The fifth character satisfies the criterion for the first state, and therefore the engine proceeds on to the second state. In this case, a match for the criterion is also present, and the engine moves on to the third state. The final character in the string matches the third state criterion, and hence a match to the pattern is made (see Figure 1-7).

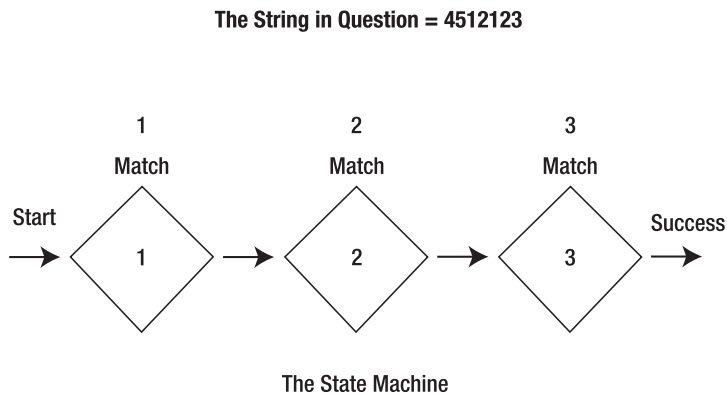
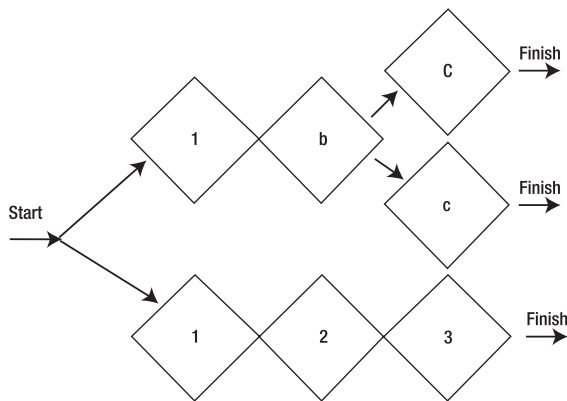


Figure 1-7. A match is made to the pattern /123/.

The previous two examples deal with a linear state machine. However, you are not limited to this type of regular expression setup. It is possible to establish alternate paths within the regular expression engine. You can set up these alternate paths by using the alternation (“or”) operator (`|`) and/or parentheses, which define subpatterns. I will cover more about the specific meanings of regular expression syntaxes in the upcoming sections “Pattern Matching,” “Quantifiers,” and “Predefined Subpatterns.” For now, consider the expression `/123|1b(c|C)/`, which specifies that the matching pattern can be 123, 1bc, or 1bC (see Figure 1-8).



The State Machine

Figure 1-8. The state machine defined by the pattern `/123|1b(c|C)/`

Note Parentheses not only define subpatterns but can also capture substrings, which I will discuss in the upcoming “Capturing Substrings” section.

As you can see, this state machine can follow multiple paths to reach the goal of a complete match. It can choose to take the top path of 123, or can choose to take one of the bottom paths of 1bc or 1bC. To get an idea of how this works, consider the string 1bc and see how the state machine would determine this to be a match. It would first find that the 1 matches the first state condition, so it would then proceed to match the next character (b) to the second state condition of the top path (2). Since this is not a match, the regular expression engine will backtrack to the location of the true state located before the “or” condition. The engine will backtrack further, in this case to the starting point, only if all the available paths are unable to provide a correct match. From this point, the regular expression engine will proceed down an alternate path, in this case the bottom one. As the engine traverses down this path, the character b is a match for the second

state of the bottom path. At this point, you have reached a second “or” condition, so the engine will check for matches along the top path first. In this case, the engine is able to match the character `c` with the required state `c`, so no further backtracking is required, and the string is considered a perfect match.

When specifying regular expression patterns, it is also beneficial to be aware of the notations `[]` and `^[^]`, since these allow you to specify ranges of characters that will serve as an acceptable match or an unacceptable one. For instance, if you had a pattern containing `[ABCDEF]` or `[A-F]`, then `A`, `B`, `C`, `D`, `E`, and `F` would all be acceptable matches. However, `a` or `G` would not be, since both are not included in the acceptable range.

Tip Perl’s regular expression patterns are case-sensitive by default. So, `A` is different from `a` unless a modifier is used to declare the expression case-insensitive. See the “Modifiers” section for more details.

If you want to specify characters that would be unacceptable, you can use the `^[^]` syntax. For example, if you want the expression to be true for any character but `A`, `B`, `C`, `D`, `E`, and `F`, you can use one of the following expressions: `^[^ABCDEF]` or `^[^A-F]`.

Pattern Matching

Now that you know how the regular expression engine functions, let’s look at how you can invoke this engine to perform pattern matches within Perl code. To perform pattern matches, you need to first acquaint yourself with the binding operators, `=~` and `!~`. The string you seek to bind (match) goes on the left, and the operator that it is going to be bound to goes on the right. You can employ three types of operators on the right side of this statement. The first is the pattern match operator, `m//`, or simply `//` (the `m` is implied and can be left out), which will test to see if the string value matches the supplied expression, such as `123 matching /123/`, as shown in Listing 1-2. The remaining two are `s///` and `tr///`, which will allow for substitution and transliteration, respectively. For now, I will focus solely on matching and discuss the other two alternatives later. When using `=~`, a value will be returned from this operation that indicates whether the regular expression operator was able to successfully match the string. The `!~` functions in an identical manner, but it checks to see if the string is unable to match the specified operator. Therefore, if a `=~` operation returns that a match was successful, the corresponding `!~` operation will not return a successful result, and vice versa. Let’s examine this a little closer by considering the simple Perl script in Listing 1-2.