

# 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](mailto: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.



# Parsing and Regular Expression Basics

**T**he 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.

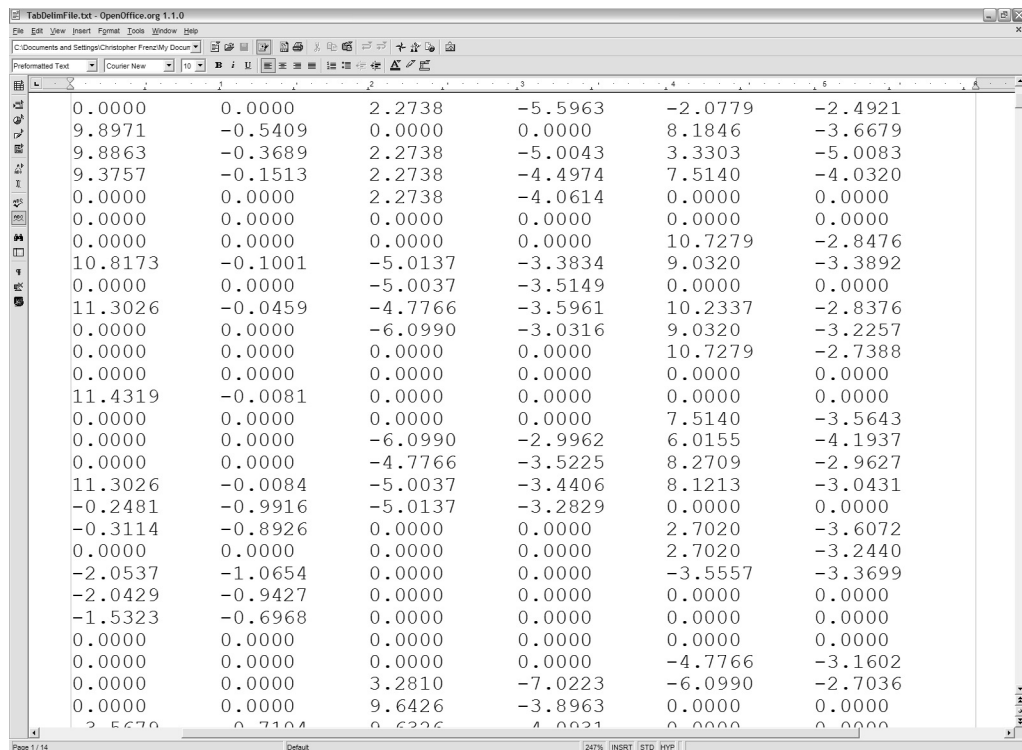
Yet, one must also consider that this generated data, while not easily accessible to all, is often put to use, resulting in the creation of new ideas to generate even more knowledge or in the creation of more efficient means of data generation. Although the old adage “knowledge is power” holds true, 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	-4.7766	-3.1602
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.

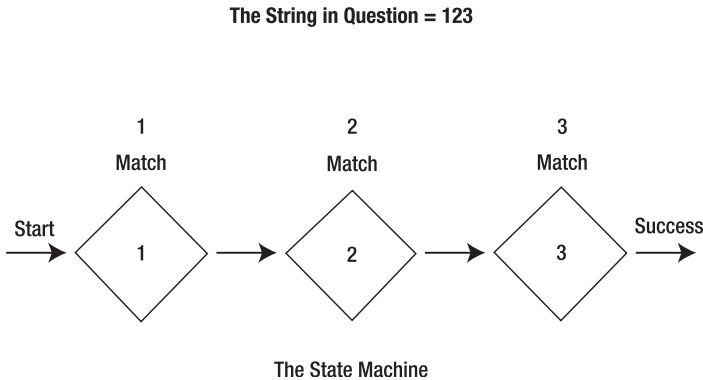
## A State Machine

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.

---



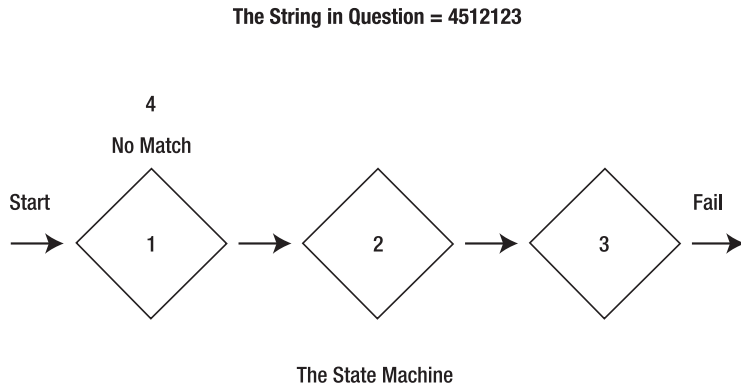
**Figure 1-2.** A state machine designed to match the pattern `/123/`

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.

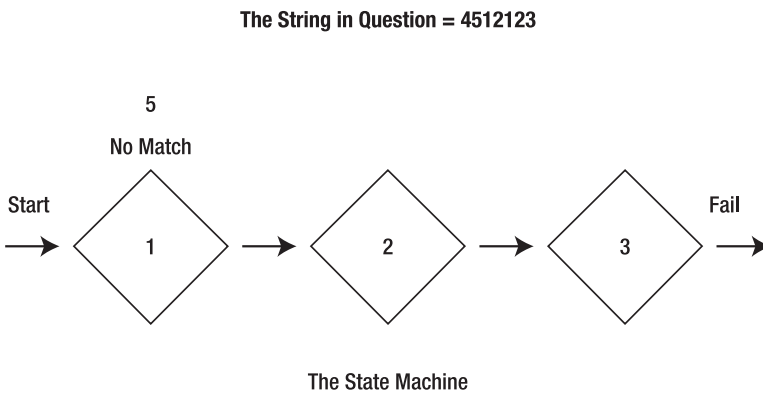
In this simple case, the string, as written, provided an exact match to the pattern. Yet, this is hardly typical in the real world, so it is important to also consider how the regular expression will operate when the character in question does not match the criterion of a particular state in the state 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).

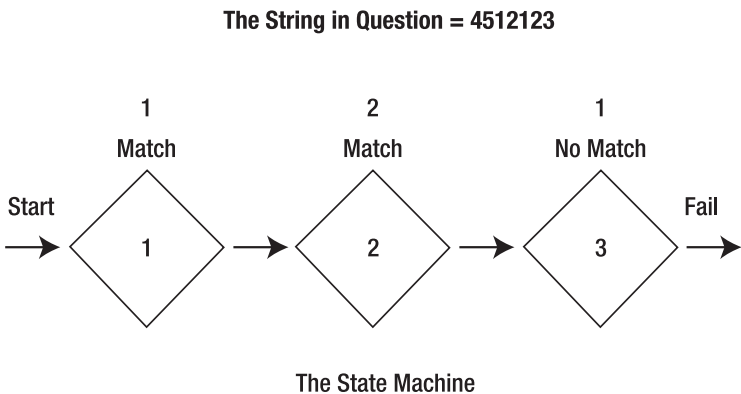
As in the first case, no match is found between criterion for the first state and the character in question (5), so the engine moves on to make a comparison beginning with the third character in the string (see Figure 1-5).



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

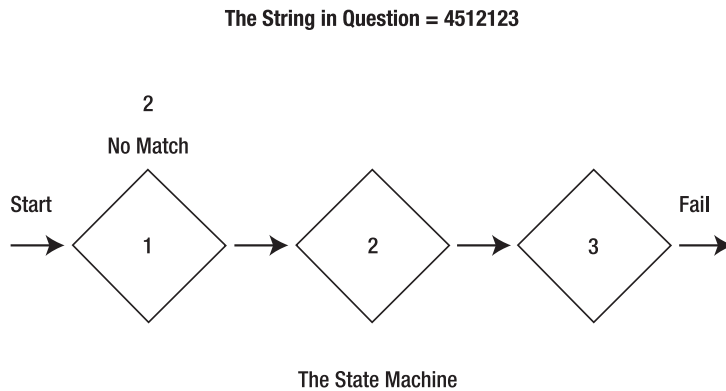


**Figure 1-4.** The second attempt at comparing the string 4512123 to the pattern /123/



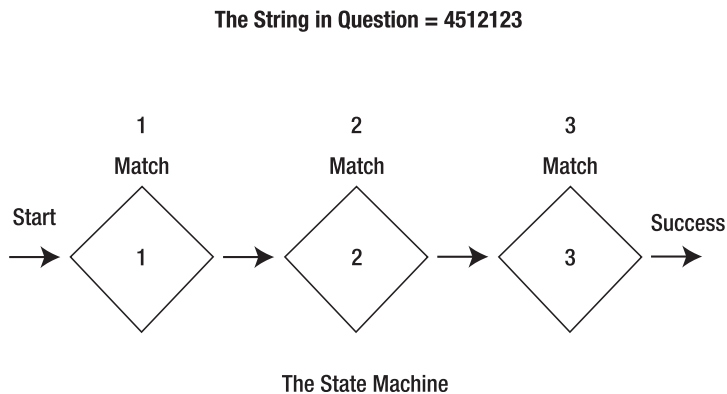
**Figure 1-5.** The third attempt at comparing the string 4512123 to the pattern /123/

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).



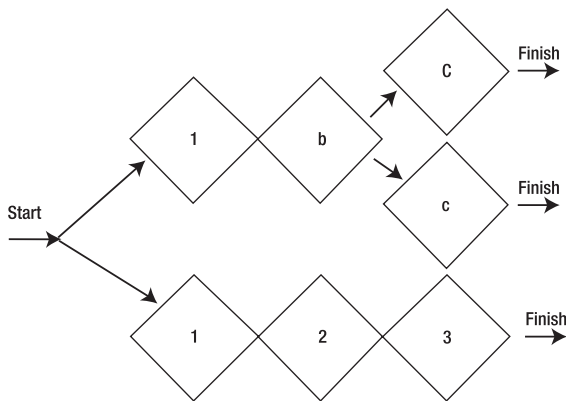
**Figure 1-6.** The fourth attempt at comparing the string 4512123 to the pattern /123/

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.

**Listing 1-2.** *Performing Some Basic Pattern Matching*

```
#!/usr/bin/perl

$string1="123";
$string2="ABC";
$pattern1="123";

if($string1=~/$pattern1/){
    print "123=123\n";
}

if($string2!~/123/){
    print "ABC does not match /123/\n";
}

if("234"=~/$pattern1|ABC/){
    print "This is 123 or ABC\n";
}
else{print "This is neither 123 nor ABC";}
```

The script begins by declaring three different scalar variables; the first two hold string values that will be matched against various regular expressions, and the third serves as storage for a regular expression pattern. Next you use a series of conditional statements to evaluate the strings against a series of regular expressions. In the first conditional, the value stored in `$string1` matches the pattern stored in `$pattern1`, so the `print` statement is able to successfully execute. In the next conditional, `$string2` does not match the supplied pattern, but the operation was conducted using the `!~` operator, which tests for mismatches, and thus this `print` statement can also execute. The third conditional does not return a match, since the string 234 does not match either alternative in the regular expression. Accordingly, in this case the `print` statement of the `else` condition will instead execute. A quick look at the output of this script confirms that the observed behavior is in agreement with what was anticipated:

```
123=123
ABC does not match /123/
This is neither 123 nor ABC
```

Operations similar to these serve as the basis of pattern matching in Perl. However, the basic types of patterns you have learned to create so far have only limited usefulness. To gain more robust pattern matching capabilities, you will now build on these basic concepts by further exploring the richness of the Perl regular expression syntax.

## Quantifiers

As you saw in the previous section, you can create a simple regular expression by simply putting the characters or the name of a variable containing the characters you seek to match between a pair of forward slashes. However, suppose you want to match the same sequence of characters multiple times. You could write out something like this to match three instances of `Yes` in a row:

```
/YesYesYes/
```

But suppose you want to match 100 instances? Typing such an expression would be quite cumbersome. Luckily, the regular expression engine allows you to use quantifiers to accomplish just such a task.

The first quantifier I will discuss takes the form of `{number}`, where `number` is the number of times you want the sequence matched. If you really wanted to match `Yes` 100 times in a row, you could do so with the following regular expression:

```
/(Yes){100}/
```

To match the whole term, putting the `Yes` in parentheses before the quantifier is important; otherwise, you would have matched `Ye` followed by 100 instances of `s`, since quantifiers operate only on the unit that is located directly before them in the pattern expression. All the quantifiers operate in a syntax similar to this (that is, the pattern followed by a quantifier); Table 1-1 summarizes some useful ones.

**Table 1-1.** *Useful Quantifiers*

Quantifier	Effect
<code>X*</code>	Zero or more <code>Xs</code> .
<code>X+</code>	One or more <code>Xs</code> .
<code>X?</code>	<code>X</code> is optional.
<code>X{5}</code>	Five <code>Xs</code> .
<code>X{5,10}</code>	From five to ten <code>Xs</code> .
<code>X{5,}</code>	Five <code>Xs</code> or more.

When using quantifiers, it is important to remember they will always produce the longest possible match unless otherwise instructed to do so. For example, consider the following string:

```
(123)123(123)
```



If you asked the regular expression engine to examine this string with an expression such as the following, you would find that the entire string was returned as a match, because `.` will match any character other than `\n` and because the string does begin and end with `(` and `)` as required:

```
/\(.*\)/
```

---

**Note** Parentheses are *metacharacters* (that is, characters with special meaning to the regular expression engine); therefore, to match either the open or close parenthesis, you must type a backslash before the character. The backslash tells the regular expression engine to treat the character as a normal character (in other words, like *a*, *b*, *c*, 1, 2, 3, and so on) and not interpret it as a metacharacter. Other metacharacters are `\`, `|`, `[`, `{`, `^`, `$`, `*`, `+`, `.`, and `?`.

---

It is important to keep in mind that the default behavior of the regular expression engine is to be greedy, which is often not wanted, since conditions such as the previous example can actually be more common than you may at first think. For example, other than with parentheses, similar issues may arise in documents if you are searching for quotes or even HTML or XML tags, since different elements and nodes often begin and end with the same tags. If you wanted only the contents of the first parentheses to be matched, you need to specify a question mark (`?`) after your quantifier. For example, if you rewrite the regular expression as follows, you find that `(123)` is returned as the match:

```
/\(.??\)/
```

Adding `?` after the quantifier allows you to control greediness and find the smallest possible match rather than the largest one.

## Predefined Subpatterns

Quantifiers are not the only things that allow you to save some time and typing. The Perl regular expression engine is also able to recognize a variety of predefined subpatterns that you can use to recognize simple but common patterns. For example, suppose you simply want to match any alphanumeric character. You can write an expression containing the pattern `[a-zA-Z0-9]`, or you can simply use the predefined pattern specified by `\w`. Table 1-2 lists other such useful subpatterns.

**Table 1-2.** *Useful Subpatterns*

Specifier	Pattern
\w	Any standard alphanumeric character or an underscore ( _ )
\W	Any nonalphanumeric character or an underscore ( _ )
\d	Any digit
\D	Any nondigit
\s	Any of \n, \r, \t, \f, and " "
\S	Any other than \n, \r, \t, \f, and " "
.	Any other than \n

These specifiers are quite common in regular expressions, especially when combined with the quantifiers listed in Table 1-1. For example, you can use \w+ to match any word, use d+ to match any series of digits, or use \s+ to match any type of whitespace. For example, if you want to split the contents of a tab-delimited text file (such as in Figure 1-1) into an array, you can easily perform this task using the `split` function as well as a regular expression involving \s+. The code for this would be as follows:

```
while (<>){
    push @Array, {split /\s+/ };
}
```

The regular expression argument provided for the `split` function tells the function where to split the input data and what elements to leave out of the resultant array. In this case, every time whitespace occurs, it signifies that the next nonwhitespace region should be a distinct element in the resultant array.

## Posix Character Classes

In the previous section, you saw the classic predefined Perl patterns, but more recent versions of Perl also support some predefined subpattern types through a set of Posix character classes. Table 1-3 summarizes these classes, and I outline their usage after the table.

**Table 1-3.** *Posix Character Classes*

Posix Class	Pattern
[ :alnum: ]	Any letter or digit
[ :alpha: ]	Any letter
[ :ascii: ]	Any character with a numeric encoding from 0 to 127
[ :cntrl: ]	Any character with a numeric encoding less than 32
[ :digit: ]	Any digit from 0 to 9 ( \d )
[ :graph: ]	Any letter, digit, or punctuation character

Posix Class	Pattern
[ :lower: ]	Any lowercase letter
[ :print: ]	Any letter, digit, punctuation, or space character
[ :punct: ]	Any punctuation character
[ :space: ]	Any space character (\s)
[ :upper: ]	Any uppercase letter
[ :word: ]	Underline or any letter or digit
[ :xdigit: ]	Any hexadecimal digit (that is, 0–9, a–f, or A–F)

---

**Note** You can use Posix characters in conjunction with Unicode text. When doing this, however, keep in mind that using a class such as [ :alpha: ] may return more results than you expect, since under Unicode there are many more letters than under ASCII. This likewise holds true for other classes that match letter and digits.

---

The usage of Posix character classes is actually similar to the previous examples where a range of characters was defined, such as [A-F], in that the characters must be enclosed in brackets. This is actually sometimes a point of confusion for individuals who are new to Posix character classes, because, as you saw in Table 1-3, all the classes already have brackets. This set of brackets is actually part of the class name, not part of the Perl regex. Thus, you actually need a second set, such as in the following regular expression, which will match any number of digits:

```
/[[:digit:]]*/
```

## Modifiers

As the name implies, modifiers allow you to alter the behavior of your pattern match in some form. Table 1-4 summarizes the available pattern modifiers.

**Table 1-4.** *Pattern Matching Modifiers*

Modifier	Function
/i	Makes insensitive to case
/m	Allows \$ and ^ to match near /n ( <b>m</b> ultiline)
/x	Allows insertion of comments and whitespace in expression
/o	Evaluates the expression variable only <b>o</b> nce
/s	Allows . to match /n ( <b>s</b> ingle line)
/g	Allows <b>g</b> lobal matching
/gc	After failed <b>g</b> lobal search, allows <b>c</b> ontinued matching

For example, under normal conditions, regular expressions are case-sensitive. Therefore, ABC is a completely different string from abc. However, with the aid of the pattern modifier /i, you could get the regular expression to behave in a case-insensitive manner. Hence, if you executed the following code, the action contained within the conditional would execute:

```
if("abc"=~ABC/i){  
    #do something  
}
```

You can use a variety of other modifiers as well. For example, as you will see in the upcoming “Assertions” section, you can use the /m modifier to alter the behavior of the ^ and \$ assertions by allowing them to match at line breaks that are internal to a string, rather than just at the beginning and ending of a string. Furthermore, as you saw earlier, the subpattern defined by . normally allows the matching of any character other than the new line metasymbol, \n. If you want to allow . to match \n as well, you simply need to add the /s modifier. In fact, when trying to match any multiline document, it is advisable to try the /s modifier first, since its usage will often result in simpler and faster executing code.

Another useful modifier that can become increasingly important when dealing with large loops or any situation where you repeatedly call the same regular expression is the /o modifier. Let’s consider the following piece of code:

```
While($string=~/$pattern/){  
    #do something  
}
```

If you executed a segment of code such as this, every time you were about to loop back through the indeterminate loop the regular expression engine would reevaluate the regular expression pattern. This is not necessarily a bad thing, because, as with any variable, the contents of the \$pattern scalar may have changed since the last iteration. However, it is also possible that you have a fixed condition. In other words, the contents of \$pattern will not change throughout the course of the script’s execution. In this case, you are wasting processing time reevaluating the contents of \$pattern on every pass. You can avoid this slowdown by adding the /o modifier to the expression:

```
While($string=~/$pattern/o){  
    #do something  
}
```

In this way, the variable will be evaluated only once; and after its evaluation, it will remain a fixed value to the regular expression engine.

---

**Note** When using the `/o` modifier, make sure you never need to change the contents of the pattern variable. Any changes you make after `/o` has been employed will not change the pattern used by the regular expression engine.

---

The `/x` modifier can also be useful when you are creating long or complex regular expressions. This modifier allows you to insert whitespace and comments into your regular expression without the whitespace or `#` being interpreted as a part of the expression. The main benefit to this modifier is that it can be used to improve the readability of your code, since you could now write `/\w+ | \d+ /x` instead of `/\w+|\d+ /`.

The `/g` modifier is also highly useful, since it allows for global matching to occur. That is, you can continue your search throughout the whole string and not just stop at the first match. I will illustrate this with a simple example from bioinformatics: DNA is made up of a series of four nucleotides specified by the letters *A*, *T*, *C*, and *G*. Scientists are often interested in determining the percentage of *G* and *C* nucleotides in a given DNA sequence, since this helps determine the thermostability of the DNA (see the following note).

---

**Note** DNA consists of two complementary strands of the nucleotides *A*, *T*, *C*, and *G*. The *A* on one strand is always bonded to a *T* on the opposing strand, and the *G* on one strand is always bonded to the *C* on the opposing strand, and vice versa. One difference is that *G* and *C* are connected by three bonds, whereas *A* and *T* only two. Consequently, DNA with more *GC* pairs is bound more strongly and is able to withstand higher temperatures, thereby increasing its thermostability.

---

Thus, I will illustrate the `/g` modifier by writing a short script that will determine the %GC content in a given sequence of DNA. Listing 1-3 shows the Perl script I will use to accomplish this.

**Listing 1-3.** *Determining %GC Content*

```
#!/usr/bin/perl;

$string="ATGCCGGGAAATTATAGCG";
$count=0;

while($string=~G|C/g){
    $count=$count+1;
}
$len=length($string);
$gc=$count/$len;
print "The DNA sequence has $gc %GC Content";
```

As you can see, you store your DNA sequence in the scalar variable `$String` and then use an indeterminate loop to step through the character content of the string. Every time you encounter a G or a C in your string, you increment your counter variable (`$Count`) by 1. After you have completed your iterations, you divide the number of Gs and Cs by the total sequence length and print your answer. For the previous DNA sequence, the output should be as follows:

```
The DNA sequence has 0.473684210526316 %GC Content
```

Under normal conditions, when the `/g` modifier fails to match any more instances of a pattern within a string, the starting position of the next search is reset back to zero. However, if you specified `/gc` instead of just `/g`, your next search would not reset back to the beginning of the string, but rather begin from the position of the last match.

## Cloistered Pattern Modifiers

In the previous section, you saw how to apply pattern modifiers to an entire regular expression. It is also possible to apply these modifiers to just a portion of a given regular expression; however, the syntax is somewhat different. The first step is to define the subpattern to which you want the modifier to apply. You accomplish this by placing the subpattern within a set of parentheses. Immediately after the open parenthesis, but before the subpattern, you add `?modifiers:`. For example, if you want to match either ABC or AbC, rather than using alternation, you write the following:

```
/A(?i:B)C/
```

To create a regular expression that allows `.` to match `/n` but only in part of the expression, you can code something like the following, which allows any character to be matched until an A is encountered:

```
/.*?A(?s:.*?)BC/
```

It then allows any character to match, including `/n`, until a BC is encountered.

---

**Note** Cloistered pattern modifiers are available only in Perl versions 5.60 and later.

---

## Assertions

Assertions are somewhat different from the topics I covered in the preceding sections on regular expressions, because unlike the other topics, assertions do not deal with characters in a string. Because of this, they are more properly referred to as *zero-width assertions*.

Assertions instead allow you to add positional considerations to your string matching capabilities. Table 1-5 summarizes the available assertions.

**Table 1-5.** *Assertions*

Assertion	Function
\A, ^	Beginning assertions
\Z, \z, \$	Ending assertions
\b	Boundary assertion
\G	Previous match assertion

### The \A and ^ Assertions

For example, if you want to match only the beginning of a string, you can employ the \A assertion. Similarly, you can also use the ^ assertion, known as the *beginning-of-line* assertion, which will match characters at the beginning of a string. When used in conjunction with the /m modifier, it will also be able to match characters after any new lines embedded within a string. Thus, if you had the regular expressions /\A123/ and /^123/m, both would be able to match the string 123456, but only /^123/m would be able to match the string abd\n123.

### The \z, \Z, and \$ Assertions

Just as there are assertions for dealing with the beginnings of lines and strings, so too are there assertions for dealing with the character sequences that end strings. The first of these assertions is the \z assertion, which will match the ending contents of a string, including any new lines. \Z works in a similar fashion; however, this assertion will not include a terminal new line character in its match, if one is present at the end of a string. The final assertion is \$, which has functionality similar to \Z, except that the /m modifier can enable this assertion to match anywhere in a string that is directly prior to a new line character. For example, /\Z321/, /\z321/, and /\$321/ would be able to match the string 654321.

### Boundary Assertions

While assertions dealing with the beginning and end of a string/line are certainly useful, assertions that allow you to deal with positions internal to a string/line are just as important. Several types of assertions can accomplish this, and the first type you will examine is the so-called boundary assertion. The \b boundary assertion allows you to perform matches at any word boundary. A word boundary can exist in two possible forms, since you have both a beginning of a word and an end. In more technical terms, the beginning

of a word boundary is defined as `\W\w`, or any nonalphanumeric character followed by any alphanumeric character. An end of a word boundary has the reverse definition. That is, it is defined by `\w\W`, or a word character followed by a nonword character. When using these assertions, you should keep in mind several considerations, however. The first is that the underscore character is a part of the `\w` subpattern, even though it is not an alphanumeric character. Furthermore, you need to be careful using this assertion if you are dealing with contractions, abbreviations, or other wordlike structures, such as Web and e-mail addresses, that have embedded nonalphanumeric characters. According to the `\w\W` or `\W\w` pattern, any of the following would contain valid boundaries:

```
"can't"
"www.apress.com"
"F.B.I."
"((1+2)*(3-4))"
user@example.com
"(555) 555-5555"
```

## The pos Function and \G Assertion

Before I discuss the remaining assertion, I will first discuss the `pos` function, since this function and the `\G` assertion are often used to similar effect. You can use the `pos` function to either return or specify the position in a string where the next matching operation will start (that is, one after the current match). To better understand this, consider the code in Listing 1-4.

### Listing 1-4. Using the pos Function

```
#!/usr/bin/perl;
$string="regular expressions are fun";
pos($string)=3;
while($string=~e/g){
    print "e at position " . (pos($string)-1). "\n";
}
```

If you execute this script, you get the following output:

```
e at position 8
e at position 12
e at position 22
```

Notice how the first e is missing from the output. This is because Listing 1-4 specified the search to begin at position 3, which is after the occurrence of the first e. Hence, when



you print the listing of the returned matches, you can see that the e in the first position was not seen by the regular expression engine.

The remaining assertion, the `\G` assertion, is a little more dynamic than the previous assertions in that it does not specify a fixed type of point where matching attempts are allowed to occur. Rather, the `\G` assertion, when used in conjunction with the `/g` modifier, will allow you to specify the position right in front of your previous match. Let's examine how this works by looking at a file containing a list of names followed by phone numbers. Listing 1-5 shows a short script that will search through the list of names until it finds a match. The script will then print the located name and the corresponding phone number.

**Listing 1-5.** *Using the `\G` Assertion*

```
#!/usr/bin/perl

($string=<<'LIST');
John (555)555-5555
Bob (234)567-8901
Mary (734)234-9873
Tom (999)999-9999
LIST

$name="Mary";
pos($string)=0;
while($string=~/$name/g){
    if($string=~/\G(\s?(?!\d{3})?[-\s.]?\d{3}[-.]?\d{4})/){
        print "$name $1";
    }
}
```

---

**Note** As mentioned earlier, parentheses are metacharacters and must be escaped in order to allow the regular expression to match them.

---

This script begins with you creating the `$string` variable and adding the list of names. Next, you define the `$name` variable as the name Mary. The next line of code is not always necessary but can be if prior matching and other types of string manipulation were previously performed on the string. You can use the `pos` function to set the starting point of the search to the starting point of the string. Finally, you can use a loop structure to search for the name Mary within your `$string` variable. Once Mary is located, you apply the `\G` assertion in the conditional statement, which will recognize and print any phone number that

is present immediately after Mary. If you execute this script, you should receive the following output:

```
Mary (734)234-9873
```

## Capturing Substrings

After looking at the previous example, you might be wondering how you were able to capture the recognized phone number in order to print it. Looking at the output and the print statement itself should give you the idea that it had something to do with the variable \$1, and indeed it did. Earlier in the chapter, I noted that parentheses could serve two purposes within Perl regular expressions. The first is to define subpatterns, and the second is to capture the substring that matches the given subpattern. These captured substrings are stored in the variables \$1, \$2, \$3, and so on. The contents of the first set of parentheses goes into \$1, the second into \$2, the third into \$3, and so on. Thus, in the previous example, by placing the phone number regular expression into parentheses, you are able to capture the phone number and print it by calling the \$1 variable.

When using nested parentheses, it is important to remember that the parentheses are given an order of precedence going from left to right, with regard to where the open parenthesis occurs. As a result, the substring is enclosed by the first open parenthesis encountered and its corresponding close parenthesis will be assigned to \$1, even if it is not the first fully complete substring to be evaluated. For example, if you instead wrote the phone number regular expression as follows, the first set of parentheses would capture the entire phone number as before:

```
=~/(\s?( \(?\d{3}\) )?)[-\s.](?\d{3}[-.]\d{4})/
```

The second set would capture the area code in \$2, and the third set would put the remainder of the phone number into \$3.

---

**Note** If you do not want to capture any values with a set of parentheses but only specify a subpattern, you can place `?:` right after `(` but before the subpattern (for example, `(?:abc)`).

---

Parentheses are not the only way to capture portions of a string after a regular expression matching operation. In addition to specifying the contents of parentheses in variables such as \$1, the regular expression engine also assigns a value to the variables \$`, \$&, and \$'. \$& is a variable that is assigned the portion of the string that the regular expression was actually able to match. \$` is assigned all the contents to the left of the match, and \$(' is assigned all the contents to the right of the match (see Table 1-6).

---

**Caution** When dealing with situations that involve large amounts of pattern matching, it may not be advisable to use `$&`, `$``, and `$'`, since if they are used once they will be repeatedly generated for every match until the Perl program terminates, which can lead to a lengthy increase in the program's execution time.

---

**Table 1-6.** *Substring Capturing Variables*

Variable	Use
<code>\$1, \$2, \$3, ...</code>	Stores captured substrings contained in parentheses
<code>\$&amp;</code>	Stores the substring that matched the regex
<code>\$`</code>	Stores the substring to the left of the matching regex
<code>\$'</code>	Stores the substring to the right of the matching regex

Let's take some time now to explore both types of capturing in greater depth by considering the medical informatics example, mentioned earlier, of mining medical literature for chemical interactions. Listing 1-6 shows a short script that will search for predefined interaction terms and then capture the names of the chemicals involved in the interaction.

**Listing 1-6.** *Capturing Substrings*

```
#!/usr/bin/perl;

($String=<<'ABOUTA');
    ChemicalA is used to treat cancer.  ChemicalA
    reacts with ChemicalB which is found in cancer
    cells.  ChemicalC inhibits ChemicalA.
ABOUTA

pos($String)=0;
while($String=~/reacts with|inhibits/ig){
    $rxn=$&;
    $left=$`;
    $right=$';
    if($left=~/(\\w+)\\s+\\z/){
        $Chem1=$1;
    }
    if($right=~/(\\w+)/){
        $Chem2=$1;
    }
    print "$Chem1 $rxn $Chem2\\n";
}
```

The script begins by searching through the text until it reaches one of the predefined interaction terms. Rather than using a dictionary-type list with numerous interaction terms, alternation of the two terms found in the text is used for simplicity. When one of the interaction terms is identified, the variable `$rxn` is set equal to this term, and `$left` and `$right` are set equal to the left and right sides of the match, respectively. Conditional statements and parentheses-based string capturing are then used to capture the word before and the word after the interaction term, since these correspond to the chemical names. It is also important to note the use of the `\z` assertion in order to match the word before the interaction term, since this word is located at the end of the `$left` string. If you run this script, you see that the output describes the interactions explained in the initial text:

```
ChemicalA reacts with ChemicalB  
ChemicalC inhibits ChemicalA
```

## Substitution

Earlier I mentioned that in addition to basic pattern matching, you can use the `=~` and `!~` operations to perform substitution. The operator for this operation is `s///`. Substitution is similar to basic pattern matching in that it will initially seek to match a specified pattern. However, once a matching pattern is identified, the substitution will replace the part of the string that matches the pattern with another string. Consider the following:

```
$String="abcdef";  
$String=~s/abc/123/;  
print $String;
```

If you execute this code, the string `a123def` will be printed. In other words, the pattern recognized by `/abc/` is replaced with `123`.

## Troubleshooting Regexes

The previous examples clearly demonstrate that regular expressions are a powerful and flexible programming tool and are thus widely applicable to a wealth of programming tasks. As you can imagine, however, all this power and flexibility can often make constructing complex regular expressions quite difficult, especially when certain positions within the expression are allowed to match multiple characters and/or character combinations. The construction of robust regular expressions is something that takes practice; but while you are gaining that experience, you should keep in mind a few common types of mistakes:

- *Make sure you choose the right wildcard:* For example, if you must have one or more of a given character, make sure to use the quantifier `+` and not `*`, since `*` will match a missing character as well.
- *Watch out for greediness:* Remember to control greediness with `?` when appropriate.
- *Make sure to check your case (for example, upper or lowercase):* For example, typing `\W` when you mean `\w` will result in the ability to match different things.
- *Watch out for metacharacters* (`\`, `(`, `,`, `|`, `[`, `{`, `^`, `$`, `*`, `+`, `.`, and `?`): If a metacharacter is part of your pattern, make sure you turn off its special meaning by prefixing it with `\`.
- *Check your | conditions carefully:* Make sure all the possible paths are appropriate.

Even with these guidelines, debugging a complex regular expression can still be a challenge, and one of the best, although time-consuming, ways to do this can be to actually draw a visual representation of how the regular expression should work, similar to that found in the state machine figures presented earlier in the chapter (Figure 1-2 through Figure 1-8). If drawing this type of schematic seems too arduous a task, you may want to consider using the `GraphViz::Regex` module.

## GraphViz::Regex

GraphViz is a graphing program developed by AT&T for the purpose of creating visual representations of structured information such as computer code (<http://www.research.att.com/sw/tools/graphviz/>). Leon Brocard wrote the GraphViz Perl module, which serves as a Perl-based interface to the GraphViz program. `GraphViz::Regex` can be useful when coding complex regular expressions, since this module is able to create visual representations of regular expressions via GraphViz. The syntax for using this module is quite straightforward and is demonstrated in the following code snippet:

```
Use GraphViz::Regex;
```

```
my $regex='((123|ab(c|C)))';
my $graph=GraphViz::Regex->new($regex);
print $graph->as_jpeg;
```

When you first employ the `GraphViz::Regex` module, you place a call to the new constructor, which requires a string of the regular expression that you seek a graphical representation of. The new method is then able to create a GraphViz object that corresponds to this representation and assigns the object to `$graph`. Lastly, you are able to print the graphical representation you created. This example displays a JPEG file, but numerous other file types are supported, including GIF, PostScript, PNG, and bitmap.

---

**Caution** The author of the module reports that there are incompatibilities between this module and Perl versions 5.005\_03 and 5.7.1.

---

---

**Tip** Another great tool for debugging regular expressions comes as a component of ActiveState's programming IDE Komodo. Komodo contains the Rx Toolkit, which allows you to enter a regular expression and a string into each of its fields and which tells you if they do or do not match as you type. This can be a rapid way to determine how well a given expression will match a given string.

---

## Using Regexp::Common

As you can imagine, certain patterns are fairly commonplace and will likely be repeatedly utilized. This is the basis behind `Regexp::Common`, which is a Perl module originally authored by Damian Conway and maintained by Abigail that provides a means of accessing a variety of regular expression patterns. Since writing regular expressions can often be tricky, you may want to check this module and see if a pattern suited to your needs is available. Table 1-7 lists all the expression pattern categories available in version 2.113 of this module.

**Table 1-7.** `Regexp::Common` *Patterns*

Pattern Types	Use
Balanced	Matches strings with parenthesized delimiters
Comment	Identifies code comments in 43 languages
Delimited	Matches delimited text
Lingua	Identifies palindromes
List	Works with lists of data
Net	Matches IPv4 and MAC Internet addresses
Number	Works with integers and reals
Profanity	Identifies obscene terms
URI	Identifies diversity of URI types
Whitespace	Matches leading and trailing whitespace
Zip	Matches ZIP codes

Although Table 1-7 provides a general idea of the different types of patterns, it is a good idea to look at the module description available at CPAN (<http://www.cpan.org/>). The module operates by generating hash values that correspond to different patterns, and these patterns are stored in the hash `%RE`. When using this module, you can access its

predefined subpatterns by referencing the scalar value of a particular hash element. So, if you want to search for Perl comments in a file, you can employ the hash value stored in `$RE{comments}{Perl}`; or, if you want to search for real numbers, you can use `$RE{num}{real}`. This two-layer hash of hash structure is fine for specifying most pattern types, but deeper layers are available in many cases. These deeper hash layers represent flags that modify the basic pattern in some form. For example, with numbers—in addition to just specifying real or integer—you can also set delimiters so that 1,234 is interpreted as a valid number pattern rather than just 1234. I will briefly cover some types of patterns, but complete coverage of every possible option could easily fill a small book on its own. I recommend you look up the module on CPAN (<http://www.cpan.org>) and refer to the descriptions of the pattern types offered by each component module.

## Regexp::Common::Balanced

This namespace generates regular expressions that are able to match sequences located between balanced parentheses or brackets. The basic syntax needed to access these regular expressions is as follows:

```
$RE{balanced}{-parens=>'()[\{\}]'}
```

The first part of this hash value refers to the basic regular expression structure needed to match text between balanced delimiters. The second part is a flag that specifies the types of parentheses you want the regular expression to recognize. In this case, it is set to work with `()`, `[]`, and `{}`. One application of such a regular expression is in the preparation of publications that contain citations, such as “(Smith et al., 1999).” An author may want to search a document for in-text citations in order to ensure they did not miss adding any to their list of references. You can easily accomplish this by passing the filename of the document to the segment of code shown in Listing 1-7.

### Listing 1-7. Pulling Out the Contents of `()` from a Document

```
#!/usr/bin/perl -w
use Regexp::Common;

while(<>){
    /$RE{balanced}{-parens=>'()' }{-keep}/
    and print "$1\n";
}
```

---

**Note** A more detailed description of the module's usage will follow in the sections “Standard Usage” and “Subroutine-Based Usage,” since each of the expression types can be accessed through code in the same manner.

---

## Regexp::Common::Comments

This module generates regular expressions that match comments inserted into computer code written in a variety of programming languages (currently 43). The syntax to call these regular expressions is as follows, where {comments} refers to the base comment matching functionality and {LANGUAGE} provides the descriptor that indicates the particular programming language:

```
$RE{comments}{LANGUAGE}
```

For example, to match Perl and C++ comments, you can use the following:

```
$RE{comments}{Perl}  
$RE{comments}{C++}
```

## Regexp::Common::Delimited

This base module provides the functionality required to match delimited strings. The syntax is similar to that shown for the Text::Balanced module:

```
$RE{delimited}{-delim=>'{'}
```

In this case, the -delim flag specifies the delimiter that the regular expression will search for and is a required flag, since the module does not have a default delimiter.

---

**Note** Table 1-8 summarizes all the Regexp::Common flags.

---

## Regexp::Common::List

The List module can match lists of data such as tab-separated lists, lists of numbers, lists of words, and so on. The type of list matched depends on the flags specified in the expression. Its syntax is as follows:

```
$RE{list}{-pat}{-sep}{-lastsep}
```



The pattern flag specifies the pattern that will correspond to each substring that is contained in the list. The pattern can be in the form of a regular expression such as `\w+` or can be another hash value created by the `Regexp::Common` module. The `-sep` flag defines a type of separator that may be present between consecutive list elements, such as a tab or a space (the default). The `-lastsep` flag specifies a separator that may be present between the last two elements in the list. By default, this value is the same as that specified by `-sep`. As an example, if you wanted to search a document for lists that were specified in the `Item A, Item B, ..., and Item N` format, you could easily identify such listings using the following expression:

```
$RE{list}{-pat}{-sep=>', '}{-lastsep=>', and '}
```

## Regexp::Common::Net

The `Net` module generates hash values that contain patterns designed to match IPv4 and MAC addresses, and the first hash key specifies which type to match. The next hash key allows you to specify whether the address will be decimal (default), hexadecimal, or octal. You can also use the `-sep` flag to specify a separator, if required. The following is a sample:

```
$RE{net}{IPv4}{hex}
```

This module comes in handy if you want to monitor the domains that different e-mails you have received originated from. This information is found in most e-mail headers in a format similar to the following:

```
from [64.12.116.134] by web51102.mail.yahoo.com via HTTP;
Mon, 29 Nov 2004 23:33:11 -0800 (PST)
```

You can easily parse this header information to find the IPv4 address `64.12.116.134` by using the following expression:

```
$RE{net}{IPv4}
```

## Regexp::Common::Number

The `Number` module can match a variety of different number types, including integers, reals, hexadecimal, octals, binaries, and even Roman numerals. The base syntax is of the following form, but you should also be aware of a diversity of flags:

```
$RE{num}{real}
```

For example, you can apply the `-base` flag to change the base of the number to something other than the default of base 10. The `-radix` flag specifies the pattern that will serve

as the decimal point in case you desire something other than the default value (.). If you are dealing with significant figures, you may find the `-places` flag useful, since it can specify the number of places after the decimal point. As in previous modules, `-sep` specifies separators; however, in this module, you can also specify the appropriate number of digits that should be present between separators using the `-group` flag. The default value for this flag is 3, so if you specified a comma (,) as your separator, your expression would be able to recognize values such as 123,456,789. The `-expon` flag specifies the pattern that will be used to specify that an exponent is present. The default value for this property is `[Ee]`.

## Universal Flags

As you saw in the previous sections, many of the base modules have their own flags, which can be used to further refine the pattern your regular expression will match (see Table 1-8). You can use two additional flags, however, with almost all base modules. These flags are the `-i` flag and the `-keep` flag. The `-i` flag makes the regular expression insensitive to alphabetic case so the expression can match both lowercase and capital letters. You can use the `-keep` flag for pattern capturing. If you specify `-keep`, the entire match to the pattern is generally stored in `$1`. In many cases, `$2`, `$3`, and other variables are also set, but these are set in a module-specific manner.

**Table 1-8.** `Regexp::Common` *Flags*

Flag	Use	Module(s)
<code>-sep</code>	Specifies a separator	Net and List
<code>-lastsep</code>	Specifies the last separator of a list	List
<code>-base</code>	For numbers, makes the base something other than base 10	Number
<code>-radix</code>	Makes a decimal point something other than .	Number
<code>-places</code>	Specifies the number of places after a decimal point	Number
<code>-group</code>	For numbers, specifies the number of digits that should be present between separators	Numbers
<code>-expon</code>	Specifies the exponent pattern	Numbers
<code>-i</code>	Makes the regular expressions case insensitive	All
<code>-keep</code>	Enables substring capturing	All

## Standard Usage

You can utilize the patterns located in the module in your source code in a couple of ways. The first of these ways is referred to as the *standard usage method* and has a syntax similar to some of the regular expressions you have already seen in that the expression is

placed between the `//` operator. The only difference is that rather than placing your own regular expression between `//`, you place one of the modules hash values. Consider the following segment of text:

```
Bob said "Hello".  James
responded "Hi, how are you".
Bob replied "Fine and you".
```

Now let's save this text to a file and execute the Perl code shown in Listing 1-8, making sure to pass the name of the file you just saved as an argument.

**Listing 1-8.** *Pulling Quotes Out of a Document*

```
#!/usr/bin/perl -w
use Regexp::Common;

while(<>){
    /$RE{delimited}{-delim=>'"}{-keep}/
    and print "$1\n";
}
```

This short piece of code will read through the contents of the file and identify all the quotes present in the text file. Since you also specified the `-keep` flag, you are able to capture the quotes and print them. Thus, the output for this script should be similar to the following:

```
"Hello"
"Hi, how are you"
"Fine and you"
```

## Subroutine-Based Usage

In addition to the standard usage, you can also access the functionality of this module through a subroutine-based interface, which allows you to perform a matching operation with a syntax similar to a procedural call. If you were to recode the previous example using this alternative syntax, it would look like Listing 1-9.

**Listing 1-9.** *Pulling Quotes Out via a Subroutine*

```
#!/usr/bin/perl -w
use Regexp::Common 'RE_ALL';

while(<>){
    $_ =~ RE_delimited(-delim=>'",-keep)
    and print "$1\n";
}
```

You should note several important things here if you choose to use this syntax instead. The first is that when you call the `Regexp::Common` module, you must append `RE_ALL` to the end of the line so Perl is able to recognize the alternative syntax. Without this, you will receive a compilation error that says the subroutines are undefined. The second noteworthy thing is that you must explicitly write `$_ =~` in order to perform the required matching operation. Lastly, you should also note that the flags are read in as arguments separated by commas. Accessing the regular expressions this way can lead to faster execution times since this method does not return objects to be interpolated but, rather, actual regular expressions.

## In-Line Matching and Substitution

I will cover these two methods together since they have similar syntax and use an object-oriented interface. In terms of basic pattern matching, they offer no real advantage other than allowing you to create code that may be somewhat more user-friendly to read; their syntax is as follows:

```
if($RE{num}{int}->matches($SomeNumber)){
    print "$SomeNumber is an Integer";
}
```

This interface allows you to easily perform substitutions on a string without changing the original string. For example:

```
$SubstitutedString=$RE{num}{real}->subs($Original=>$Substitution);
```

In this case, `$SubstitutedString` is a new string that is going to be assigned the value of the `$Original` string with all substitutions already made, and the `$Substitution` string specifies the string that is going to be put in place of the characters that were able to match the pattern.

## Creating Your Own Expressions

The `Regexp::Common` module does not limit you to just the patterns that come with it. You also have the ability to create your own regular expressions, at run time, for use within the `Regexp::Common` module. For example, `Regexp::Common` does not yet support phone numbers, so let's begin to create a `Regexp::Common` phone number entry (see Listing 1-10).

**Listing 1-10.** *Creating Your Own Regexp::Common Expression*

```
#!/usr/bin/perl -w
use Regexp::Common qw /pattern/;

pattern name=>[qw(phone)],
    create=>q/(?k:\s?(\\d{3}\\))[-\\s.](\\d{3}[-.]\\d{4})?)/;

while(<>){
    /$RE{phone}{-keep}/ and print "$1\\n";
}
```

---

**Note** You may have noticed that the pattern contains the sequence of characters `?k:` in it. Under normal circumstances, capturing through parentheses is not preserved in `Regexp::Common`, since capturing parentheses are processed out. The `?k:` sequence tells the module not to process out these parentheses when the `-keep` flag is present. This is why you were able to print phone numbers by using `$1` in the previous example.

---

To begin, you must first tell Perl you are going to utilize the pattern subroutine of the `Regexp::Common` module. Next, you must create a `name` argument that will specify the name of the pattern and any flags it may take. In this case, the pattern is named `phone`. If you want to add additional names and/or flags, you can specify them as follows:

```
pattern name=[qw(phone book -flag)]
```

This specifies an entry of `$RE{phone}{book}{-flag}`.

After you name your pattern, you must next specify a value for the `create` argument. This argument is the only other required argument and can take either a string that is to be returned as a pattern (as previously) or a reference to a subroutine that will create the pattern. Also, two optional arguments also take subroutine references. These arguments are `match` and `subs`, and the provided subroutine will dictate what occurs when the methods `match` and `subs`, the matching and substitution methods (respectively), are called. Lastly, one more optional argument, `version`, can be assigned a Perl version number.

If the version of Perl is older than the supplied argument, the script will not run and a fatal error will be returned.

## Summary

This chapter covered how to syntactically construct regular expressions and how you can call upon these expressions within your Perl scripts. Furthermore, I discussed the roles of the different quantifiers, assertions, and predefined subpatterns, as well as how best to debug regular expressions. Lastly, the chapter covered how the Perl module `Regexp::Common` works and how you can utilize it to locate elements of interest.

Now that you have an idea of how you can use regular expressions to match, and hence identify, portions of strings, you are more prepared to tackle the topics of tokens and grammars in greater depth as you delve into the next chapter. Chapter 2 will introduce you to the idea of generative grammars by covering the Chomsky hierarchy of grammars. The upcoming chapter will also demonstrate how you can use Perl code in conjunction with a grammar to generate sentences that comply with the rules specified in the grammar.