5

Regular Expressions

"11:15. Restate my assumptions:

1. *Mathematics is the language of nature.*
2. *Everything around us can be represented and understood through numbers.*
3. *If you graph these numbers, patterns emerge. Therefore: There are patterns everywhere in nature."*

- Max Cohen in Pi*, 1998*

Whether or not you agree that Max's assumptions give rise to his conclusion is your own opinion, but his case is much easier to follow in the field of computers – there are certainly patterns everywhere in programming.

Regular expressions allow us look for patterns in our data. So far we've been limited to checking a single value against that of a scalar variable or the contents of an array or hash. By using the rules outlined in this chapter, we can use that one single value (or pattern) to describe what we're looking for in more general terms: we can check that every sentence in a file begins with a capital letter and ends with a full stop, find out how many times James Bond's name is mentioned in 'Goldfinger', or learn if there are any repeated sequences of numbers in the decimal representation of greater than five in length.

However, regular expressions are a very big area – they're one of the most powerful features of Perl. We're going to break our treatment of them up into six sections:

- Basic patterns
- Special characters to use
- Quantifiers, anchors and memorizing patterns
- Matching, substituting, and transforming text using patterns
- Backtracking
- A quick look at some simple pitfalls

Generally speaking, if you want to ask perl something about a piece of text, regular expressions are going to be your first port of call – however, there's probably one simple question burning in your head…

What Are They?

The term "Regular Expression" (now commonly abbreviated to "RegExp" or even "RE") simply refers to a pattern that follows the rules of syntax outlined in the rest of this chapter. Regular expressions are not limited to perl – Unix utilities such as `sed` and `egrep` use the same notation for finding patterns in text. So why aren't they just called 'search patterns' or something less obscure?

Well, the actual phrase itself originates from the mid-fifties when a mathematician called Stephen Kleene developed a notation for manipulating 'regular sets'. Perl's regular expressions have grown and grown beyond the original notation and have significantly extended the original system, but some of Kleene's notation remains, and the name has stuck.

Patterns

History lessons aside, it's all about identifying patterns in text. So what constitutes a pattern? And how do you compare it against something?

The simplest pattern is a word – a simple sequence of characters – and we may, for example, want to ask perl whether a certain string contains that word. Now, we can do this with the techniques we have already seen: We want to split the string into separate words, and then test to see if each word is the one we're looking for. Here's how we might do that:

```
#!/usr/bin/perl
# match1.plx
use warnings;
use strict;

my $found = 0;
$_ = "Nobody wants to hurt you... 'cept, I do hurt people sometimes, Case.";

my $sought = "people";

foreach my $word (split) {
   if ($word eq $sought) {
      $found = 1;
      last;
   }
}

if ($found) {
   print "Hooray! Found the word 'people'\n";
}
```

Sure enough the program returns success:

```
>perl match1.plx
Hooray! Found the word 'people'
>
```

But that's messy! It's complicated, and it's slow to boot! Worse still, the `split` function (which breaks each of our lines up into a list of 'words' – we'll see more of this, later on in the chapter) actually **keeps** all the punctuation – the string '`you`' wouldn't be found in the above, whereas '`you...`' would. This looks like a hard problem, but it should be easy. Perl was designed to make easy tasks easy and hard things possible, so there should be a better way to do this. This is how it looks using a regular expression:

```
#!/usr/bin/perl
# match1.plx
use warnings;
use strict;

$_ =    "Nobody wants to hurt you... 'cept, I do hurt people sometimes, Case.";

if ($_ =~ /people/) {
   print "Hooray! Found the word 'people'\n";
}
```

This is much, much easier and yeilds the same result. We place the text we want to find between forward slashes – that's the regular expression part – that's our pattern, what we're trying to match. We also need to tell perl which particular string we're looking for in that pattern. We do this with the =~ operator. This returns 1 if the pattern match was successful (in our case, whether the character sequence 'people' was found in the string) and the undefined value if it wasn't.

Before we go on to more complicated patterns, let's just have a quick look at that syntax. As we noted previously, a lot of Perl's operations take $_ as a default argument, and regular expressions are one such operation. Since we have the text we want to test in $_, we don't need to use the =~ operator to 'bind' the pattern to another string. We could write the above even more simply:

```
$_ = "Nobody wants to hurt you... 'cept, I do hurt people sometimes, Case.";
if (/people/) {
   print "Hooray! Found the word 'people'\n";
}
```

Alternatively, we might want to test for the pattern not matching – the word not being found. Obviously, we could say `unless (/people/)`, but if the text we're looking at isn't in $_, we may also use the negative form of that =~ operator, which is `!~`. For example:

```
#!/usr/bin/perl
# nomatch.plx
use warnings;
use strict;

my $gibson =
   "Nobody wants to hurt you... 'cept, I do hurt people sometimes, Case.";

if ($gibson !~ /fish/) {
   print "There are no fish in William Gibson.\n";
}
```

True to form, for cyberpunk books that don't regularly involve fish, we get the result.

```
>perl nomatch.plx
There are no fish in William Gibson.
>
```

Literal text is the simplest regular expression of all to look for, but we needn't look for just the one word – we could look for any particular phrase. However, we need to make sure that we exactly match *all* the characters: words (with correct capitalization), numbers, punctuation, and even whitespace:

```
#!/usr/bin/perl
# match2.plx
use warnings;
use strict;

$_ = "Nobody wants to hurt you... 'cept, I do hurt people sometimes, Case.";

if (/I do/) {
   print "'I do' is in that string.\n";
}

if (/sometimes Case/) {
   print "'sometimes Case' matched.\n";
}
```

Let's run this program and see what happens:

```
>perl match2.plx
'I do' is in that string.
>
```

The other string didn't match, even though those two words are there. This is because everything in a regular expression has to match the string, from start to finish: first "sometimes", then a space, then "Case". In `$_`, there was a comma before the space, so it didn't match exactly. Similarly, spaces inside the pattern are significant:

```
#!/usr/bin/perl
# match3.plx
use warnings;
use strict;

my $test1 = "The dog is in the kennel";
my $test2 = "The sheepdog is in the field";

if ($test1 =~ / dog/) {
   print "This dog's at home.\n";
}

if ($test2 =~ / dog/) {
   print "This dog's at work.\n";
}
```

This will only find the first dog, as perl was looking for a space followed by the three letters, 'dog':

```
>perl match3.plx
This dog's at home.
>
```

So, for the moment, it looks like we shall have to specify our patterns with absolute precision. As another example, look at this:

```
#!/usr/bin/perl
# match4.plx
use warnings;
use strict;
```

```
$_ = "Nobody wants to hurt you... 'cept, I do hurt people sometimes, Case.";
if (/case/) {
   print "I guess it's just the way I'm made.\n";
} else {
   print "Case? Where are you, Case?\n";
}
```

```
> perl match4.plx
Case? Where are you, Case?
>
```

Hmm, no match. Why not? Because we asked for a small 'c' when we had a big 'C' – regexps are (if you'll pardon the pun) case-sensitive. We can get around this by asking perl to compare insensitively, and we do this by putting an 'i' (for 'insensitive') after the closing slash. If we alter the code above as follows:

```
if (/case/i) {
   print "I guess it's just the way I'm made.\n";
} else {
   print "Case? Where are you, Case?\n";
}
```

then we find him:

```
>perl match4.plx
I guess it's just the way I'm made.
>
```

This 'i' is one of several **modifiers** that we can add to the end of the regular expression to change its behavior slightly. We'll see more of them later on.

Interpolation

Regular expressions work a little like double-quoted strings; variables and metacharacters are interpolated. This allows us to store patterns in variables and determine what we are matching when we run the program – we don't need to have them hard-coded in:

Try it out – Pattern Tester

This program will ask the user for a pattern and then test to see if it matches our string. We can use this throughout the chapter to help us test the various different styles of pattern we'll be looking at:

```
#!/usr/bin/perl
# matchtest.plx
use warnings;
use strict;

$_ = q("I wonder what the Entish is for 'yes' and 'no'," he thought.);
# Tolkien, Lord of the Rings

print "Enter some text to find: ";
my $pattern = <STDIN>;
chomp($pattern);
```

```
if (/$pattern/) {
   print "The text matches the pattern '$pattern'.\n";
} else {
   print "'$pattern' was not found.\n";
}
```

Now we can test out a few things:

> **perl matchtest.plx**
Enter some text to find: **wonder**
The text matches the pattern 'wonder'.

> **perl matchtest.plx**
Enter some text to find: **entish**
'entish' was not found.

> **perl matchtest.plx**
Enter some text to find: **hough**
The text matches the pattern 'hough'.

> **perl matchtest.plx**
Enter some text to find: **and 'no',**
The text matches the pattern 'and 'no''.

Pretty straightforward, and I'm sure you could all spot those not in $_ as well.

How It Works

`matchtest.plx` has its basis in the three lines:

```
my $pattern = <STDIN>;
chomp($pattern);

if (/$pattern/) {
```

We're taking a line of text from the user. Then, since it will end in a new line, and we don't necessarily want to find a new line in our pattern, we `chomp` it away. Now we do our test.

Since we're not using the `=~` operator, the test will be looking at the variable `$_`. The regular expression is `/$pattern/`, and just like the double-quoted string `"$pattern"`, the variable `$pattern` is interpolated. Hence, the regular expression is purely and simply whatever the user typed in, once we've got rid of the new line.

Escaping Special Characters

Of course, regular expressions can be more than just words and spaces. The rest of this chapter is going to be about the various ways we can specify more advanced matches – where portions of the match are allowed to be one of a number of characters, or where the match must occur at a certain position in the string. To do this, we'll be describing the special meanings given to certain characters – called **metacharacters** – and look at what these meanings are and what sort of things we can express with them.

At this stage, we might not want to use their special meanings – we may want to literally match the characters themselves. As you've already seen with double-quoted strings, we can use a backslash to escape these characters' special meanings. Hence, if you want to match '. . .' in the above text, you need your pattern to say '\ . \ . \ .'. For example:

```
> perl matchtest.plx
Enter some text to find: Ent+
The text matches the pattern 'Ent+'.

> perl matchtest.plx
Enter some text to find: Ent\+
'Ent\+' was not found.
```

We'll see later why the first one matched – due to the special meaning of +.

> **These are the characters that are given special meaning within a regular expression, which you will need to backslash if you want to use literally:**
>
> **. * ? + [ ] ( ) { } ^ $ | **
>
> **Any other characters automatically assume their literal meanings.**

You can also turn off the special meanings using the escape sequence `\Q`. After perl sees `\Q`, the 14 special characters above will automatically assume their ordinary, literal meanings. This remains the case until perl sees either `\E` or the end of the pattern.

For instance, if we wanted to adapt our `matchtest` program just to look for literal strings, instead of regular expressions, we could change it to look like this:

```
    if (/\Q$pattern\E/) {
```

Now the meaning of + is turned off:

```
> perl matchtest.plx
Enter some text to find: Ent+
'Ent+' was not found.
>
```

Note that all `\Q` does is turn off the regular expression magic of those 14 characters above – it doesn't stop, for example, variable interpolation.

> *Don't forget to change this back again: We'll be using `matchtest.plx` throughout the chapter, to demonstrate the regular expressions we look at. We'll need that magic fully functional!*

Anchors

So far, our patterns have all tried to find a match anywhere in the string. The first way we'll extend our regular expressions is by dictating to perl where the match must occur. We can say 'these characters must match the beginning of the string' or 'this text must be at the end of the string'. We do this by **anchor**ing the match to either end.

The two anchors we have are ^, which appears at the beginning of the pattern anchor a match to the beginning of the string, and $ which appears at the end of the pattern and anchors it to the end of the string. So, to see if our quotation ends in a full stop – and remember that the full stop is a special character – we say something like this:

```
>perl matchtest.plx
Enter some text to find: \.$
The text matches the pattern '\.$'.
```

That's a full stop (which we've escaped to prevent it being treated as a special character) and a dollar sign at the end of our pattern – to show that this must be the end of the string.

Try, if you can, to get into the habit of reading out regular expressions in English. Break them into pieces and say what each piece does. Also remember to say that each piece must immediately follow the other in the string in order to match. For instance, the above could be read 'match a full stop immediately followed by the end of the string'.

If you can get into this habit, you'll find that reading and understanding regular expressions becomes a lot easier, and you'll be able to 'translate' back into Perl more naturally as well.

Here's another example: do we have a capital I at the beginning of the string?

```
> perl matchtest.plx
Enter some text to find: ^I
'^I' was not found.
>
```

We use ^ to mean 'beginning of the string', followed by an I. In our case, though, the character at the beginning of the string is a ", so our pattern does not match. If you know that what you're looking for can only occur at the beginning or the end of the match, it's extremely efficient to use anchors. Instead of searching through the whole string to see whether the match succeeded, perl only needs to look at a small portion and can give up immediately if even the first character does not match.

Let's see one more example of this, where we'll combine looking for matches with looking through the lines in a file:

Try it out : Rhyming Dictionary

Imagine yourself as a poor poet. In fact, not just poor, but downright bad – so bad, you can't even think of a rhyme for 'pink'. So, what do you do? You do what every sensible poet does in this situation, and you write the following Perl program:

```
#!/usr/bin/perl
# rhyming.plx
use warnings;
use strict;

my $syllable = "ink";
while (<>) {
   print if /$syllable$/;
}
```

We can now feed it a file of words, and find those that end in 'ink':

```
>perl rhyming.plx wordlist.txt
blink
bobolink
brink
chink
clink
>
```

For a really thorough result, you'll need to use a file containing every word in the dictionary – be prepared to wait though if you do! For the sake of the example however, any text-based file will do (though it'll help if it's in English). A bobolink, in case you're wondering, is a migratory American songbird, otherwise known as a ricebird or reedbird.

How It Works

With the loops and tests we learned in the last chapter, this program is really very easy:

```
while (<>) {
   print if /$syllable$/;
}
```

We've not looked at file access yet, so you may not be familiar with the `while(<>){...}` construction used here. In this example it opens a file that's been specified on the command line, and loops through it, one line at a time, feeding each one into the special variable `$_` – this is what we'll be matching.

Once each line of the file has been fed into `$_`, we test to see if it matches the pattern, which is our syllable, 'ink', anchored to the end of the line (with `$`). If so, we print it out.

The important thing to note here is that perl treats the 'ink' as the last thing on the line, even though there is a new line at the end of `$_`. Regular expressions typically ignore the last new line in a string – we'll look at this behavior in more detail later.

Shortcuts and Options

All this is all very well if we know exactly what it is we're trying to find, but finding patterns means more than just locating exact pieces of text. We may want to find a three-digit number, the first word on the line, four or more letters all in capitals, and so on.

We can begin to do this using **character classes** – these aren't just single characters, but something that signifies that any one of a *set* of characters is acceptable. To specify this, we put the characters we consider acceptable inside square brackets. Let's go back to our `matchtest` program, using the same test string:

```
$_ = q("I wonder what the Entish is for 'yes' and 'no'," he thought.);
```

```
> perl matchtest.plx
Enter some text to find: w[aoi]nder
The text matches the pattern 'w[aoi]nder'.
>
```

What have we done? We've tested whether the string contains a 'w', followed by either an 'a', an 'o', or an 'i', followed by 'nder'; in effect, we're looking for either of 'wander', 'wonder', or 'winder'. Since the string contains 'wonder', the pattern is matched.

Conversely, we can say that everything is acceptable **except** a given sequence of characters – we can 'negate the character class'. To do this, the character class should start with a `^`, like so:

```
> perl matchtest.plx
Enter some text to find: th[^eo]
'th[^eo]' was not found.
>
```

So, we're looking for 'th' followed by something that is neither an 'e' or an 'o'. But all we have is 'the' and 'thought', so this pattern does not match.

If the characters you wish to match form a sequence in the character set you're using – ASCII or Unicode, depending on your perl version – you can use a hyphen to specify a range of characters, rather than spelling out the entire range. For instance, the numerals can be represented by the character class `[0-9]`. A lower case letter can be matched with `[a-z]`. Are there any numbers in our quote?

```
> perl matchtest.plx
Enter some text to find: [0-9]
'[0-9]' was not found.
>
```

You can use one or more of these ranges alongside other characters in a character class, so long as they stay inside the brackets. If you wanted to match a digit and then a letter from 'A' to 'F', you would say `[0-9][A-F]`. However, to match a single hexadecimal digit, you would write `[0-9A-F]` or `[0-9A-Fa-f]` if you wished to include lower-case letters.

Some character classes are going to come up again and again: the digits, the letters, and the various types of whitespace. Perl provides us with some neat shortcuts for these. Here are the most common ones, and what they represent:

Shortcut	Expansion	Description
`\d`	`[0-9]`	Digits 0 to 9.
`\w`	`[0-9A-Za-z_]`	A 'word' character allowable in a Perl variable name.
`\s`	`[ \t\n\r]`	A whitespace character that is, a space, a tab, a newline or a return.

also, the negative forms of the above:

Shortcut	Expansion	Description
`\D`	`[^0-9]`	Any non-digit.
`\W`	`[^0-9A-Za-z_]`	A non-'word' character.
`\S`	`[^ \t\n\r]`	A non-blank character.

So, if we wanted to see if there was a five-letter word in the sentence, you might think we could do this:

```
> perl matchtest.plx
Enter some text to find: \w\w\w\w\w
The text matches the pattern '\w\w\w\w\w'.
>
```

But that's not right – there are no five-letter words in the sentence! The problem is, we've only asked for five letters in a row, and any word with **at least** five letters contains five in a row will match that pattern. We actually matched 'wonde', which was the first possible series of five letters in a row. To actually get a five-letter word, we might consider deciding that the word must appear in the middle of the sentence, that is, between two spaces:

```
> perl matchtest.plx
Enter some text to find: \s\w\w\w\w\w\s
'\s\w\w\w\w\w\s' was not found.
>
```

Word Boundaries

The problem with that is, when we're looking at text, words aren't always between two spaces. They can be followed by or preceded by punctuation, or appear at the beginning or end of a string, or otherwise next to non-word characters. To help us properly search for words in these cases, Perl provides the special `\b` metacharacter. The interesting thing about `\b` is that it doesn't actually match any character in particular. Rather, it matches the point between something that isn't a word character (either `\W` or one of the ends of the string) and something that is (a word character), hence `\b` for **b**oundary. So, for example, to look for one-letter words:

```
> perl matchtest.plx
Enter some text to find: \s\w\s
'\s\w\s' was not found.

> perl matchtest.plx
Enter some text to find: \b\w\b
The text matches the pattern '\b\w\b'.
```

As the I was preceded by a quotation mark, a space wouldn't match it – but a word boundary does the job. Later, we'll learn how to tell perl how many repetitions of a character or group of characters we want to match without spelling it out directly.

What, then, if we wanted to match anything at all? You might consider something like `[\w\W]` or `[\s\S]`, for instance. Actually, this is quite a common operation, so Perl provides an easy way of specifying it – a full stop. What about an 'r' followed by two characters – any two characters – and then a 'h'?

```
> perl matchtest.plx
Enter some text to find: r..h
The text matches the pattern 'r..h'.
>
```

Is there anything after the full stop?

```
> perl matchtest.plx
Enter some text to find: \..
'\..' was not found.
>
```

What's that? One backslashed full stop to mean a full stop, then a plain one to mean 'anything at all'.

Posix and Unicode Classes

Perl 5.6.0 introduced a few more character classes into the mix – first, those defined by the POSIX (Portable Operating Systems Interface) standard, which are therefore present in a number of other applications. The more common character classes here are:

Shortcut	Expansion	Description
`[[:alpha:]]`	`[a-zA-Z]`	An alphabetic character.
`[[:alnum:]]`	`[0-9A-Za-z]`	An alphabetic or numeric character.
`[[:digit:]]`	`\d`	A digit, 0-9.
`[[:lower:]]`	`[a-z]`	A lower case letter.
`[[:upper:]]`	`[A-Z]`	An upper case letter.
`[[:punct:]]`	`[!"#$%&'()*+,-./:;<=>?@\[\\\]^_`{|}~]`	A punctuation character – note the escaped characters `[`, `\`, and `]`.

The Unicode standard also defines 'properties', which apply to some characters. For instance, the '`IsUpper`' property can be used to match any upper-case character, in whichever language or alphabet. If you know the property you are trying to match, you can use the syntax `\p{}` to match it, for instance, the upper-case character is `\p{IsUpper}`.

Alternatives

Instead of giving a series of acceptable characters, you may want to say 'match either this or that'. The 'either-or' operator in a regular expression is the same as the bitwise 'or' operator, `|`. So, to match either 'yes' or 'maybe' in our example, we could say this:

```
> perl matchtest.plx
Enter some text to find: yes|maybe
The text matches the pattern 'yes|maybe'.
>
```

That's either 'yes' or 'maybe'. But what if we wanted either 'yes' or 'yet'? To get alternatives on part of an expression, we need to group the options. In a regular expression, grouping is always done with parentheses:

```
> perl matchtest.plx
Enter some text to find: ye(s|t)
The text matches the pattern 'ye(s|t)'.
>
```

If we have forgotten the parentheses, we would have tried to match either 'yes' or 't'. In this case, we'd still get a positive match, but it wouldn't be doing what we want – we'd get a match for any string with a 't' in it, whether the words 'yes' or 'yet' were there or not.

You can match either 'this' or 'that' or 'the other' by adding more alternatives:

```
> perl matchtest.plx
Enter some text to find: (this)|(that)|(the other)
'(this)|(that)|(the other)' was not found.
>
```

However, in this case, it's more efficient to separate out the common elements:

```
> perl matchtest.plx
Enter some text to find: th(is|at|e other)
'th(is|at|e other)' was not found.
```

You can also nest alternatives. Say you want to match one of these patterns:

- 'the' followed by whitespace or a letter,
- 'or'

You might put something like this:

```
> perl matchtest.plx
Enter some text to find: (the(\s|[a-z]))|or
The text matches the pattern '(the(\s|[a-z]))|or'.
>
```

It looks fearsome, but break it down into its components. Our two alternatives are:

- `the(\s|[a-z])`
- `or`

The second part is easy, while the first contains 'the' followed by two alternatives: `\s` and `[a-z]`. Hence 'either "the" followed by either a whitespace or a lower case letter, or "or"'. We can, in fact, tidy this up a little, by replacing `(\s|[a-z])` with the less cluttered `[\sa-z]`.

```
> perl matchtest.plx
Enter some text to find: (the[\sa-z])|or
The text matches the pattern '(the[\sa-z])|or'.
>
```

Repetition

We've now moved from matching a specific character to a more general type of character – when we don't know (or don't care) exactly what the character will be. Now we're going to see what happens when we want to talk about a more general quantity of characters: more than three digits in a row; two to four capital letters, and so on. The metacharacters that we use to deal with a number of characters in a row are called **quantifiers**.

Indefinite Repetition

The easiest of these is the question mark. It should suggest uncertainty – something may be there, or it may not. That's exactly what it does: stating that the immediately preceding character(s) – or metacharacter(s) – may appear once, or not at all. It's a good way of saying that a particular character or group is optional. To match the word 'he or she', you can put:

```
> perl matchtest.plx
Enter some text to find: \bs?he\b
The text matches the pattern '\bs?he\b'.
>
```

To make a series of characters (or metacharacters) optional, group them in parentheses as before. Did he say 'what the Entish is' or 'what the Entish word is'? Either will do:

```
> perl matchtest.plx
Enter some text to find: what the Entish (word )?is
The text matches the pattern 'what the Entish (word )?is'.
>
```

Notice that we had to put the space inside the group: otherwise we end up with two spaces between 'Entish' and 'is', whereas our text only has one:

```
> perl matchtest.plx
Enter some text to find: what the Entish (word)? is
'what the Entish (word)? is' was not found.
>
```

As well as matching something one or zero times, you can match something one or more times. We do this with the plus sign – to match an entire word without specifying how long it should be, you can say:

```
> perl matchtest.plx
Enter some text to find: \b\w+\b
The text matches the pattern '\b\w+\b'.
>
```

In this case, we match the first available word – I.

If, on the other hand, you have something which may be there any number of times but might not be there at all – zero or one or many – you need what's called 'Kleene's star': the * quantifier. So, to find a capital letter after any – but possibly no – spaces at the start of the string, what would you do? The start of the string, then any number of whitespace characters, then a capital:

```
> perl matchtest.plx
Enter some text to find: ^\s*[A-Z]
'^\s*[A-Z]' was not found.
>
```

Of course, our test string begins with a quote, so the above pattern won't match, but, sure enough, if you take away that first quote, the pattern will match fine.

Let's review the three qualifiers:

`/bea?t/`	Matches either 'beat' or 'bet'
`/bea+t/`	Matches 'beat', 'beaat', 'beaaat'…
`/bea*t/`	Matches 'bet', 'beat', 'beaat'…

Novice Perl programmers tend to go to town on combinations of dot and star, and the results often surprise them, particularly when it comes to searching-and-replacing. We'll explain the rules of the regular expression matcher shortly, but bear the following in mind:

A regular expression should hardly ever start or finish with a starred character.

You should also consider the fact that `.*` and `.+` in the middle of a regular expression will match as much of your string as they possibly can. We'll look more at this 'greedy' behavior later on.

Well-Defined Repetition

If you want to be more precise about how many times a character or roups of characters might be repeated, you can specify the maximum and minimum number of repeats in curly brackets. '2 or 3 spaces' can be written as follows:

```
> perl matchtest.plx
Enter some text to find: \s{2,3}
'\s{2,3}' was not found.
>
```

So we have no doubled or trebled spaces in our string. Notice how we construct that – the minimum, a comma, and the maximum, all inside braces. Omitting either the maximum or the minimum signifies 'or more' and 'or fewer' respectively. For example, `{2,}` denotes '2 or more', while `{,3}` is '3 or fewer'. In these cases, the same warnings apply as for the star operator.

Finally, you can specify exactly how many things are to be in a row by simply putting that number inside the curly brackets. Here's the five-letter-word example tidied up a little:

```
> perl matchtest.plx
Enter some text to find: \b\w{5}\b
'\b\w{5}\b' was not found.
>
```

Summary Table

To refresh your memory, here are the various metacharacters we've seen so far:

Metacharacter	Meaning
`[abc]`	any one of the characters `a`, `b`, or `c`.
`[^abc]`	any one character other than `a`, `b`, or `c`.

Table continued on following page

Metacharacter	Meaning
`[a-z]`	any one ASCII character between `a` and `z`.
`\d \D`	a digit; a non-digit.
`\w \W`	a 'word' character; a non-'word' character.
`\s \S`	a whitespace character; a non-whitespace character.
`\b`	the boundary between a `\w` character and a `\W` character.
`.`	any character (apart from a new line).
`(abc)`	the phrase '`abc`' as a group.
`?`	preceding character or group may be present 0 or 1 times.
`+`	preceding character or group is present 1 or more times.
`*`	preceding character or group may be present 0 or more times.
`{x,y}`	preceding character or group is present between *x* and *y* times.
`{,y}`	preceding character or group is present at most *y* times.
`{x,}`	preceding character or group is present at least *x* times.
`{x}`	preceding character or group is present *x* times.

Backreferences

What if we want to know what a certain regular expression matched? It was easy when we were matching literal strings: we knew that 'Case' was going to match those four letters and nothing else. But now, what matches? If we have `/\w{3}/`, which three word characters are getting matched?

Perl has a series of special variables in which it stores anything that's matched with a group in parentheses. Each time it sees a set of parentheses, it copies the matched text inside into a numbered variable – the first matched group goes in `$1`, the second group in `$2`, and so on. By looking at these variables, which we call the **backreference** variables, we can see what triggered various parts of our match, and we can also extract portions of the data for later use.

First, though, let's rewrite our test program so that we can see what's in those variables:

Try it out : A Second Pattern Tester

```
#!/usr/bin/perl
# matchtest2.plx
use warnings;
use strict;

$_ = '1: A silly sentence (495,a) *BUT* one which will be useful. (3)';

print "Enter a regular expression: ";
my $pattern = <STDIN>;
chomp($pattern);
```

```
if (/$pattern/) {
    print "The text matches the pattern '$pattern'.\n";
    print "\$1 is '$1'\n" if defined $1;
    print "\$2 is '$2'\n" if defined $2;
    print "\$3 is '$3'\n" if defined $3;
    print "\$4 is '$4'\n" if defined $4;
    print "\$5 is '$5'\n" if defined $5;
} else {
    print "'$pattern' was not found.\n";
}
```

Note that we use a backslash to escape the first 'dollar' symbol in each `print` *statement, thus displaying the actual symbol, while leaving the second in each to display the contents of the appropriate variable.*

We've got our special variables in place, and we've got a new sentence to do our matching on. Let's see what's been happening:

```
> perl matchtest2.plx
Enter a regular expression: ([a-z]+)
The text matches the pattern '([a-z]+)'.
$1 is 'silly'

> perl matchtest2.plx
Enter a regular expression: (\w+)
The text matches the pattern '(\w+)'.
$1 is '1'

> perl matchtest2.plx
Enter a regular expression: ([a-z]+)(.*)([a-z]+)
The text matches the pattern '([a-z]+)(.*)([a-z]+)'.
$1 is 'silly'
$2 is ' sentence (495,a) *BUT* one which will be usefu'
$3 is 'l'

> perl matchtest2.plx
Enter a regular expression: e(\w|n\w+)
The text matches the pattern 'e(\w|n\w+)'.
$1 is 'n'
```

How It Works

By printing out what's in each of the groups, we can see exactly what caused perl to start and stop matching, and when. If we look carefully at these results, we'll find that they can tell us a great deal about how perl handles regular expressions.

How the Engine Works

We've now seen most of the syntax behind regular expression matching and plenty of examples of it in action. The code that does all the matching is called perl's 'regular expression engine'. You might now be wondering about the exact rules applied by this engine when determining whether or not a piece of text matches. And how much of it matches what. From what our examples have shown us, let us make some deductions about the engine's operation.

Our first expression, `([a-z]+)` plucked out a set of one-or-more lower-case letters. The first such set that perl came across was '`silly`'. The next character after '`y`' was a space, and so no longer matched the expression.

- **Rule one**: Once the engine starts matching, it will keep matching a character at a time for as long as it can. Once it sees something that doesn't match, however, it has to stop. In this example, it can never get beyond a character that is not a lower case letter. It has to stop as soon as it encounters one.

Next, we looked for a series of word characters, using (`\w+`). The engine started looking at the beginning of the string and found one, '1'. The next character was not a word character (it was a colon), and so the engine had to stop.

- **Rule two**: Unlike me, the engine is **eager**. It's eager to start work and eager to finish, and it starts matching as soon as possible in the string; if the first character doesn't match, try and start matching from the second. Then take every opportunity to finish as quickly as possible.

Then we tried this: `([a-z]+)(.*)([a-z]+)`. The result we got with this was a little strange. Let's look at it again:

```
> perl matchtest2.plx
Enter a regular expression: ([a-z]+)(.*)([a-z]+)
The text matches the pattern '([a-z]+)(.*)([a-z]+)'.
$1 is 'silly'
$2 is ' sentence (495,a) *BUT* one which will be usefu'
$3 is 'l'
>
```

Our first group was the same as what matched before – nothing new there. When we could no longer match lower case letters, we switched to matching anything we could. Now, this *could* take up the rest of the string, but that wouldn't allow a match for the third group. We have to leave at least one lower-case letter.

So, the engine started to reverse back along the string, giving characters up one by one. It gave up the closing bracket, the 3, then the opening bracket, and so on, until we got to the first thing that would satisfy all the groups and let the match go ahead – namely a lower-case letter: the 'l' at the end of 'useful'.

From this, we can draw up the third rule:

- **Rule three**: Like me, in this case, the engine is **greedy**. If you use the `+` or `*` operators, they will try and steal as much of the string as possible. If the rest of the expression does not match, it grudgingly gives up a character at a time and tries to match again, in order to find the fullest possible match.

We can turn a greedy match into a non-greedy match by putting the `?` operator after either the plus or star. For instance, let's turn this example into a non-greedy version: `([a-z]+)(.*?)([a-z]+)`. This gives us an entirely different result:

```
> perl matchtest2.plx
Enter a regular expression: ([a-z]+)(.*?)([a-z]+)
The text matches the pattern '([a-z]+)(.*?)([a-z]+)'.
$1 is 'silly'
$2 is ' '
$3 is 'sentence'
>
```

Now we've shut off rule three, rule two takes over. The smallest possible match for the second group was a single space. First, it tried to get nothing at all, but then the third group would be faced with a space. This wouldn't match. So, we grudgingly accept the space and try and finish again. This time the third group has some lower case letters, and that can match as well.

What if we turn off greediness in all three groups, and say this: `([a-z]+?)(.*?)([a-z]+?)`

```
> perl matchtest2.plx
Enter a regular expression: ([a-z]+?)(.*?)([a-z]+?)
The text matches the pattern '([a-z]+?)(.*?)([a-z]+?)'.
$1 is 's'
$2 is ''
$3 is 'i'
>
```

What about this? Well, the smallest possible match for the first group is the 's' of silly. We asked it to find one character or more, and so the smallest it could find was one. The second group actually matched no characters at all. This left the third group facing an 'i', which it took to complete the match.

Our last example included an alternation:

```
> perl matchtest2.plx
Enter a regular expression: e(\w|n\w+)
The text matches the pattern 'e(\w|n\w+)'.
$1 is 'n'
>
```

The engine took the first branch of the alternation and matched a single character, even though the second branch would actually satisfy greed. This leads us onto the fourth rule:

- **Rule four**: Again like me, the regular expression engine **hates decisions**. If there are two branches, it will always choose the first one, even though the second one might allow it to gain a longer match.

To summarize:

> **The regular expression engine starts as soon as it can, grabs as much as it can, then tries to finish as soon as it can, while taking the first decision available to it.**

Working with RegExps

Now that we've matched a string, what do we do with it? Well, sometimes it's just useful to know whether a string contains a given pattern or not. However, a lot of the time we're going to be doing search-and-replace operations on text. We'll explain how to do that here. We'll also cover some of the more advanced areas of dealing with regular expressions.

Substitution

Now we know all about matching text, substitution is very easy. Why? Because all of the clever things are in the 'search' part, rather than the 'replace': all the character classes, quantifiers and so on only make sense when matching. You can't substitute, say, a word with any number of digits. So, all we need to do is take the 'old' text, Our match, and tell perl what we want to replace it with. This we do with the `s///` operator.

The `s` is for 'substitute' – between the first two slashes, we put our regular expression as before. Before the final slash, we put our text replacement. Just as with matching, we can use the `=~` operator to apply it to a certain string. If this is not given, it applies to the default variable `$_`:

```
#!/usr/bin/perl
# subst1.plx
use warnings;
use strict;

$_ = "Awake! Awake! Fear, Fire, Foes! Awake! Fire, Foes! Awake!";
# Tolkien, Lord of the Rings

s/Foes/Flee/;
print $_,"\n";
```

```
>perl subst1.plx
Awake! Awake! Fear, Fire, Flee! Awake! Fire, Foes! Awake!
>
```

Here we have substituted the first occurrence of 'Foes' with the word 'Flee'. Had we wanted to change every occurrence, we would have needed to use another modifier. Just as the `/i` modifier for matching case-insensitively, the `/g` modifier on a substitution acts **g**lobally:

```
#!/usr/bin/perl
# subst1.plx
use warnings;
use strict;

$_ = "Awake! Awake! Fear, Fire, Foes! Awake! Fire, Foes! Awake!";
# Tolkien, Lord of the Rings

s/Foes/Flee/g;
print $_,"\n";
```

```
> perl subst1.plx
Awake! Awake! Fear, Fire, Flee! Awake! Fire, Flee! Awake!
>
```

Like the left-hand side of the substitution, the right-hand side also works like a double-quoted string and is thus subject to variable interpolation. One useful thing, though, is that we can use the backreference variables we collected during the match on the right hand side. So, for instance, to swap the first two words in a string, we would say something like this:

```
#!/usr/bin/perl
# subst2.plx
use warnings;
use strict;

$_ = "there are two major products that come out of Berkeley: LSD and UNIX";
# Jeremy Anderson

s/(\w+)\s+(\w+)/$2 $1/;
print $_, "?\n";
```

```
>perl subst2.plx
are there two major products that come out of Berkeley: LSD and UNIX?
>
```

What would happen if we tried doing that globally? Well, let's do it and see:

```
#!/usr/bin/perl
# subst2.plx
use warnings;
use strict;

$_ = "there are two major products that come out of Berkeley: LSD and UNIX";
# Jeremy Anderson

s/(\w+)\s+(\w+)/$2 $1/g;
print $_, "?\n";
```

```
>perl subst2.plx
are there major two that products out come Berkeley of: and LSD UNIX?
>
```

Here, every word in a pair is swapped with its neighbor. When processing a global match, perl always starts where the previous match left off.

Changing Delimiters

You may have noticed that `//` and `s///` looks like `q//` and `qq//`. Well, just like `q//` and `qq//`, we can change the delimiters when matching and substituting to increase the readability of our regular expressions. The same rules apply: Any non-word character can be the delimiter, and paired delimiters such as `<>`, `()`, `{}`, and `[]` may be used – with two provisos.

First, if you change the delimiters on `//`, you must put an `m` in front of it. (m for 'match'). This is so that perl can still recognize it as a regular expression, rather than a block or comment or anything else.

Second, if you use paired delimiters with substitution, you must use two pairs:

```
s/old text/new text/g;
```

becomes:

```
s{old text}{new text}g;
```

You may, however, leave spaces or new lines between the pairs for the sake of clarity:

```
s{old text}
 {new text}g;
```

The prime example of when you would want to do this is when you are dealing with file paths, which contain a lot of slashes. If you are, for instance, moving files on your Unix system from `/usr/local/share/` to `/usr/share/`, you may want to munge the file names like this:

```
s/\/usr\/local\/share\//\/usr\/share\//g;
```

However, it's far easier and far less ugly to change the delimiters in this case:

```
s#/usr/local/share/#/usr/share/#g;
```

Modifiers

We've already seen the `/i` modifier used to indicate that a match should be case insensitive. We've also seen the `/g` modifier to apply a substitution. What other modifiers are there?

- `/m` – treat the string as multiple lines. Normally, `^` and `$` match the very start and very end of the string. If the `/m` modifier is in play, then they will match the starts and ends of individual lines (separated by `\n`). For example, given the string: `"one\ntwo"`, the pattern `/^two$/` will not match, but `/^two$/m` will.
- `/s` – treat the string as a single line. Normally, `.` does not match a new line character; when `/s` is given, then it will.
- `/g` – as well as globally replacing in a substitution, allows us to match multiple times. When using this modifier, placing the `\G` anchor at the beginning of the regexp will anchor it to the end point of the last match.
- `/x` – allow the use of whitespace and comments inside a match.

Regular expressions can get quite fiendish to read at times. The `/x` modifier is one way to stop them becoming so. For instance, if you're matching a string in a log file that contains a time, followed by a computer name in square brackets, then a message, the expression you'll create to extract the information may easily end up looking like this:

```
# Time in $1, machine name in $2, text in $3
/^([0-2]\d:[0-5]\d:[0-5]\d)\s+\[([^\]]+)\]\s+(.*)$/
```

However, if you use the /x modifier, you can stretch it out as follows:

```
/^
(         # First group: time
   [0-2]\d
   :
   [0-5]\d
   :
   [0-5]\d
   )
\s+
\[      # Square bracket
   (         # Second group: machine name
   [^\]]+   # Anything that isn't a square bracket
   )
\]      # End square bracket


\s+
   (         # Third group: everything else
   .*
   )
$/x
```

Another way to tidy this up is to put each of the groups into variables and interpolate them:

```
my $time_re = '([0-2]\d:[0-5]\d:[0-5]\d)';
my $host_re = '\[[^\]]+)\]';
my $mess_re = '(.*)';

/^$time_re\s+$host_re\s+$mess_re$/;
```

Split

We briefly saw `split` earlier on in the chapter, where we used it to break up a string into a list of words. In fact, we only saw it in a very simple form. Strictly speaking, it was a bit of a cheat to use it at all. We didn't see it then, but `split` was actually using a regular expression to do its stuff!

Using `split` on its own is equivalent to saying:

```
split /\s+/, $_;
```

which breaks the default string `$_` into a **list** of substrings, using whitespace as a delimiter. However, we can also specify our own regular expression: perl goes through the string, breaking it whenever the regexp matches. The delimiter itself is thrown away.

For instance, on the UNIX operating system, configuration files are sometimes a list of fields separated by colons. A sample line from the password file looks like this:

```
kake:x:10018:10020::/home/kake:/bin/bash
```

To get at each field, we can split when we see a colon:

```
#!/usr/bin/perl
# split.plx
use warnings;
use strict;

my $passwd = "kake:x:10018:10020::/home/kake:/bin/bash";
my @fields = split /:/, $passwd;
print "Login name : $fields[0]\n";
print "User ID : $fields[2]\n";
print "Home directory : $fields[5]\n";
```

```
>perl split.plx
Login name : kake
User ID : 10018
Home directory : /home/kake
>
```

Note that the fifth field has been left empty. Perl will recognize this as an empty field, and the numbering used for the following entries takes account of this. So `$fields[5]` returns `/home/kake`, as we'd otherwise expect. Be careful though – if the line you are splitting contains empty fields at the end, they will get dropped.

Join

To do the exact opposite, we can use the `join` operator. This takes a specified delimiter and interposes it between the elements of a specified array. For example:

```
#!/usr/bin/perl
# join.plx
use warnings;
use strict;

my $passwd = "kake:x:10018:10020::/home/kake:/bin/bash";
my @fields = split /:/, $passwd;
print "Login name : $fields[0]\n";
print "User ID : $fields[2]\n";
print "Home directory : $fields[5]\n";

my $passwd2 = join "#", @fields;
print "Original password : $passwd\n";
print "New password :      $passwd2\n";
```

```
>perl join.plx
Login name : kake
User ID : 10018
Home directory : /home/kake
Original password : kake:x:10018:10020::/home/kake:/bin/bash
New password :      kake#x#10018#10020##/home/kake#/bin/bash
>
```

Transliteration

While we're looking at regular expressions, we should briefly consider another operator. While it's not directly associated with regexps, the transliteration operator has a lot in common with them and adds a very useful facility to the matching and substitution techniques we've already seen.

What this does is to correlate the characters in its two arguments, one by one, and use these pairings to substitute individual characters in the referenced string. It uses the syntax `tr/one/two/` and (as with the matching and substitution operators) references the special variable `$_` unless otherwise specified with `=~` or `!~`. In this case, it replaces all the 'o's in the referenced string with 't's, all the 'n's with 'w's, and all the 'e's with 'o's.

Let's say you wanted to replace, for some reason, all the numbers in a string with letters. You might say something like this:

```
$string =~ tr/0123456789/abcdefghij/;
```

This would turn, say, "2011064" into "cabbage". You can use ranges in transliteration but not in any of the character classes. We could write the above as:

```
$string =~ tr/0-9/a-j/;
```

The return value of this operator is, by default, the number of characters matched with those in the first argument. You can therefore use the transliteration operator to count the number of occurrences of certain characters. For example, to count the number of vowels in a string, you can use:

```
my $vowels = $string =~ tr/aeiou//;
```

Note that this will not actually substitute any of the vowels in the variable `$string`. As the second argument is blank, there is no correlation, so no substitution occurs. However, the transliteration operator can take the `/d` modifier, which *will* delete occurrences on the left that do not have a correlating character on the right. So, to get rid of all spaces in a string quickly, you could use this line:

```
$string =~ tr/ //d;
```

Common Blunders

There are a few common mistakes people tend to make when writing regexps. We've already seen that `/a*b*c*/` will happily match any string at all, since it matches each letter zero times. What else can go wrong?

- **Forgetting To Group**
  `/Bam{2}/` will match 'Bamm', while `/(Bam){2}/` will match 'BamBam', so be careful when choosing which one to use. The same goes for alternation: `/Simple|on/` will match 'Simple' and 'on', while `/Sim(ple|on)/` will match both 'Simple' and 'Simon' Group each option separately.
- **Getting The Anchors Wrong**
  `^` goes at the beginning, `$` goes at the end. A dollar anywhere else in the string makes perl try and interpolate a variable.

- **Forgetting To Escape Special Characters**.
  Do you want them to have a special meaning? These are the characters to be careful of: `. * ? + [ ] ( ) { } ^ $ | and of course \ itself.`
- **Not Counting from Zero**
  The first entry in an array is given the index zero.
- **Counting from Zero**
  I know, I know! All along I've been telling you that computers start counting from zero. Nevertheless, there's always the odd exception – the first backreference is `$1`. Don't blame Perl though – it took this behavior from a language called `awk` which used `$1` as the first reference variable.

More Advanced Topics

We've not actually plumbed the depths of the regular expression language syntax – Perl has a habit of adding wilder and more bizarre features to it on a regular basis. All of the more off-the-wall extensions begin with a question mark in a group – this is supposed to make you stop and ask yourself: 'Do I really want to do this?'

Some of these are experimental and may change from perl version to version (and may soon disappear altogether), but there are others that aren't so tricky. Some of these are extremely useful, so let's dive in!

Inline Comments

We've already seen how we can use the `/x` modifier to add comments and whitespace to our regular expressions. We can also do this with the `(?#)` pattern:

```
/^Today's (?# This is ignored, by the way)date:/
```

Unfortunately, there's no way to have parentheses inside these comments, since perl closes the comment as soon as it sees a closing bracket. If you want to have longer or more detailed comments, you should consider using the `/x` modifier instead.

Inline Modifiers

If you are reading patterns from a file or constructing them from inside your code, you have no way of adding a modifier to the end of the regular expression operator. For example:

```
#!/usr/bin/perl
# inline.plx
use warnings;
use strict;

my $string = "There's more than One Way to do it!";

print "Enter a test expression: ";
my $pat = <STDIN>;
chomp($pat);

if ($string =~ /$pat/) {
print "Congratulations! '$pat' matches the sample string.\n";
} else {
print "Sorry. No match found for '$pat'";
}
```

If we run this and momentarily forgot how our sample string had been capitalized, we might get this:

```
>perl inline.plx
Enter a test expression: one way to do it!
Sorry. No match found for 'one way to do it!'
>
```

So how can we make this case-insensitive? The solution is to use an inline modifier, the syntax for which is `(?i)`. This will make the enclosing group match case-insensitively. Therefore we have:

```
>perl inline.plx
Enter a test expression: (?i)one way to do it!
Congratulations! '(?i)one way to do it!' matches the sample string.
>
```

If, conversely, you have a modifier in place that you temporarily want to get rid of, you can say, for example, `(?-i)` to turn it off. If we have this:

```
/There's More Than ((?-i)One Way) To Do It!/i;
```

the words 'One Way' alone are matched case-sensitively.

Note that you can also inline the `/m`, `/s`, and `/x` modifiers in the same way.

Grouping without Backreferences

Parentheses perform the function of grouping and populating the backreference variables. If you have a portion of your match in parentheses, it will, if successful, be placed in one of the numbered variables. However, there may be times when you only want to use brackets for grouping. For example, you're expecting the first backreference to contain something important, but there may be some preceding text in the way. You could have something like this:

```
/(X-)?Topic: (\w+)/;
```

You can't be certain whether your first defined backreference is going to end up in `$1` or `$2` – it depends on whether the '`X-`' part is present or not. For example, if we tried to match the string "Topic: the weather", we'd find that `$1` was left undefined. If we'd tried to do something with its contents, we'd get the warning:

```
Use of uninitialized value in concatenation
```

Now that's not necessarily a problem here. After all, we'll find our word in `$2` whether or not there's anything preceding `"Topic: "`. Surely we can just be careful not to use `$1`?

But what if there's more than one optional field? Say we had an expression that left all but the 2nd and 6th groups optional. We then have to look in $2 for our first word and $6 for our second, while $1, $3, $4, and $5 are left undefined. This really isn't good programming style and *is* asking for trouble! We really shouldn't backreference fields if we don't need to.

We can resolve this problem very simply, by adding the characters `?:` like this:

```
/(?:X-)?Topic: (\w+)/;
```

This ensures that the first set of brackets will now group only and not fill a backreference variable. Our word will always be put into `$1`.

Lookaheads and Lookbehinds

Sometimes you may want to say something along the lines of 'substitute the word "fish" with "cream", but only if the next word is "cake".' You can do this very simply by saying:

```
s/fish cake/cream cake/
```

What does this do? The regular expression engine scans a referenced string, looking for a match on "fish cake" On finding one, it substitutes the text "cream cake". Not too bad – it does the job. In this case it's not too big a deal that it has to substitute five characters from each match with five *identical* characters from the substitution string. It's not hard to see how this sort of inefficiency could really start to bog a program down if we used substitutions excessively.

What we want is a way of putting an assertion into the match – a 'match the text *only if* the next word is "cake"' clause – without actually matching the assertion itself. Having matched "fish", we really just want to *look ahead*, to see if it says " cake" (and give the match a thumbs-up if it does), then forget about "cake" altogether.

In life, that's not so easy. Fortunately in Perl we have an operator for just this sort of thing:

```
/fish(?= cake)/
```

will match exactly what we want – it looks for "fish", does a positive lookahead on " cake", and matches "fish" only if that succeeds. For example:

```
#!/usr/bin/perl
# look1.plx
use warnings;
use strict;

$_ = "fish cake and fish pie";
print "Our original order was ", $_, "\n";

s/fish(?= cake)/cream/;
print "Actually, make that ", $_, " instead.\n";
```

will return

```
>perl look1.plx
Our original order was fish cake and fish pie
Actually, make that cream cake and fish pie instead.
>
```

We can also look ahead negatively, by using an exclamation mark instead of the equals sign:

```
/fish(?! cake)/
```

which will match "fish" only if the following word is *not* " cake". If we adapt `look1.plx` like so:

```
#!/usr/bin/perl
# look2.plx
use warnings;
use strict;

$_ = "fish cake and fish pie";
print "Our original order was ", $_, "\n";

s/fish(?! cake)/cream/;
print "Actually, make that ", $_, " instead.\n";
```

then sure enough, it's "fish pie" that gets matched this time and not "fish cake".

```
>perl look2.plx
Our original order was fish cake and fish pie
Actually, make that fish cake and cream pie instead.
>
```

Lookaheads are very powerful as you'll soon discover if you experiment a little, particularly when you start to use less specific expressions (using metacharacters) with them.

However, we may also wish to look at the text preceding a matched pattern. We therefore have a similar pair of **lookbehind** operators. We now use the < sign to point 'behind' the match, matching "cake" only if "fish" *precedes* it. So to find all those boring old fish cakes, we use:

```
/(?<=fish )cake/
```

but to find all the cream cakes and chocolate cakes, do this:

```
/(?<!fish )cake/
```

Let's have fish and chips instead of our fish cakes and cream doughnuts instead of cream cakes:

```
#!/usr/bin/perl
# look3.plx
use warnings;
use strict;

$_ = "fish cake and cream cake";
print "Our original order was ", $_, "\n";

s/(?<=fish )cake/and chips/;
print "No, wait. I'll have ", $_, " instead\n";

s/(?<!fish )cake/slices/;
print "Actually, make that ", $_, ", will you?\n";
```

```
>perl look3.plx
Our original order was fish cake and cream cake
No, wait. I'll have fish and chips and cream cake instead
Actually, make that fish and chips and cream slices, will you?
>
```

One very important thing to note about lookbehind assertions is that they can only handle fixed-width expressions. So while you *can* use most of the metacharacters, indeterminate quantifiers like `.`, `?`, and `*` aren't allowed.

Backreferences (again)

Finally, in our tour of regular expressions, let's look again at backreferences. Suppose you want to find any repeated words in a string. How would you do it? You might think about doing this:

```
if (/\b(\w+) $1\b/) {
   print "Repeated word: $1\n";
}
```

Except, this doesn't work, because `$1` is only set when the match is complete. In fact, if you have warnings turned on, you'll be alerted to the fact that `$1` is undefined every time. In order to match while still inside the regular expression, you need to use the following syntax:

```
if (/\b(\w+) \1\b/) {
   print "Repeated word: $1\n";
}
```

However, when you're replacing, you'll get a warning if you try and use the `\<number>` syntax on the wrong side. It'll work, but you'll be told "`\1 better written as $1`".

Summary

Regular expressions are quite possibly the most powerful means at your disposal of looking for patterns in text, extracting sub-patterns and replacing portions of text. They're the basis of any text shuffling you do in Perl, and they should be your first port of call when you need to do some string manipulation.

In this chapter, we've seen how to match simple text, different classes of text, and then different amounts of text. We've also seen how to provide alternative matches, how to refer back to portions of the match, and how to substitute and transliterate text.

The key to learning and understanding regular expressions is to be able to break them down into their component parts and unravel the language, translating it piecewise into English. Once you can fluently read out the intention of a complex regular expression, you're well on your way to creating powerful matches of your own.

You can find a summary of regular expression syntax in Appendix A. Section 6 of the Perl FAQ (at www.perl.com) contains a good selection of regexp hints and tricks.

Exercises

1. Write out English descriptions of the following regular expressions, and describe what the operations actually do:

```
$var =~ /(\w+)$/

$code !~ /^#/

s/#{2,}/#/g
```

2. Using the contents of the gettysburg.txt file (provided in the download for Chapter 6), use regular expressions to do the following, and print out the result. (Tip: use a here-document to store the text in your file):

   a. Count the number of occurences of the word 'we'.

   b. Reformat the text, so that each sentences is displayed as a separate paragraph.

   c. Check that there are no multiple spaces in the text, replacing any with single spaces.

3. When we use groups, the // operator returns a list of all the text strings that have been matched. Modify our example program matchtest2.plx, so that it produces its output from this list, rather than using special variables.

4. If we want to sort a list of words into alphabetical order, one simple and quite effective way is to write a program that performs a 'bubble sort': working through the whole list, it compares each pair of consecutive words; if it finds them in the wrong order, it swaps them over. On reaching the end of the list it repeats the process – unless the previous scan didn't yield any swaps, in which case the list is already properly ordered. Use regular expressions along with the other techniques you've seen so far, and write this program so that it will work with a list of words separated by newline characters. One small hint – the pos() function may come in useful here. You can use this to adjust the position of the \G boundary, for example: pos($var) = 10 will set it just after the tenth character in $var. A subsequent global search will therefore start from this point.