



## C H A P T E R 1 0

---

# *Regular expressions*

- 10.1 The basic components 184
- 10.2 The character class 188
- 10.3 Greedy quantifiers: take what you can get 191
- 10.4 Non-greedy quantifiers: take what you need 192
- 10.5 Simple anchors 193
- 10.6 Grouping, capturing, and backreferences 195
- 10.7 Other anchors: lookahead and lookbehind 198
- 10.8 Exercises 201

Regular expressions (singular *regex* or plural *regexen*) are a powerful device for general text processing. Perl's particular brand of regular expression language is more powerful and rich in features than almost any other flavor of regular expression. Unfortunately, this additional power comes with a price of additional complexity. In this chapter, we will examine Perl's regular expression language in some detail, starting with just the basics and moving on to discuss most of the features of the *regex* language in recent versions of Perl. (We won't, however, cover some new features in Perl's version 5.005 that are at an experimental stage.) We will explore the match and substitution operators, as well as a few additional string functions in chapter 11.

## 10.1 The basic components

Figure 10.1 provides a table of most elements of Perl's *regex* language that we will consider in this chapter. These are presented here simply for reference purposes and to give you an idea of what's to come. By the end of this chapter, you should have a good idea of what all this funky punctuation actually means. Even then, as with any language, it will take some time and practice before these elements become familiar. (For a full look at Perl's regular expressions, see the *perlre* pod-page).

<p style="text-align: center;"><b>Quantifiers</b></p> <p>* + ? {n} {min,} {min, max}</p> <p>*? +? ?? {n}? {min,}? {min,max}?</p>		<p style="text-align: center;"><b>Grouping</b></p> <p>(...)</p> <p>(?...)</p>
<p><b>Character Class</b></p> <p>[...]</p> <p>[^...]</p>	<p style="text-align: center;"><b>Anchors</b></p> <p>^ \$ \A \Z \G \b \B</p>	<p style="text-align: center;"><b>Alternation</b></p> <p> </p>
<p style="text-align: center;"><b>Double Quote Sequences</b></p> <p>\b \t \n \r \f \a \e \num \xnum \cchar</p> <p style="text-align: center;">\L \l \U \u \Q \E</p>		<p style="text-align: center;"><b>Lookahead Lookbehind</b></p> <p>(?!...)</p> <p>(?=...)</p> <p>(?&lt;!...)</p> <p>(?&lt;=...)</p>
<p><b>Backreferences</b></p> <p>\1, \2, \3 ...</p>	<p style="text-align: center;"><b>Class Sequences</b></p> <p>\s \w \d \S \W \D</p>	<p style="text-align: center;"><b>Dot</b></p> <p>.</p>

**Figure 10.1** Perl's regular expression elements

We took a limited look at using regular expressions and Perl's pattern matching and substitution operators back in chapter 6. In this chapter, we will start afresh and

present most of the regular expression language elements available in recent versions of Perl. To begin with, we will consider five simple concepts that we are already familiar with from chapter 6 and then use these to demonstrate how pattern matching works. Then we'll go on to consider the more sophisticated regex elements.

The five simple concepts we need to begin with are:

- 1 Concatenation: This is an implicit concept that simply allows us to combine two or more simple patterns into a larger pattern. For example, `m/£/` is a simple pattern, and `m/o/` is a simple pattern; the pattern `m/£oo/` is a concatenation of three simple patterns to form a larger pattern.
- 2 Alternation: The meta-character (`|`) is the alternation operator that allows us to specify a choice among two or more patterns. For example, `m/£oo|bar/` will match if it finds `£oo` or `bar` starting at any position in the target string.
- 3 Grouping: Parentheses provide a grouping mechanism that allows us to create subpatterns that can be treated as a unit. The pattern `m/£(u|oo)bar` is a concatenation of the elements `£`, `(u|oo)`, and `bar`. The parentheses also provide a limiting scope for the alternation operator. Parentheses are also used to capture matched substrings into the special `$1`, `$2`, `$3` . . . variables as explained in chapter 6 (we will present a special form of grouping parentheses that do not capture matched text in section 10.6).
- 4 Iteration: The star operator (`*`) is an iterative or repetitive construct that matches zero-or-more of the previous pattern element, if it is a single character or a subexpression delimited by parentheses. For example, `m/£o*/` will match an `£` followed zero-or-more `o` characters. The pattern `m/(£oo)*/` will match zero-or-more strings of `£oo`. When I say zero-or-more, the behavior is to match as many as possible while letting the remainder of the regex match.
- 5 Dot: The dot (`.`) is a wildcard operator that will match any single character except a newline character. (This operator can be modified with the `/s` modifier, as we will see in the next chapter.) Thus, the pattern `m/£.o/` will match an `£` followed by any character followed by an `o`.

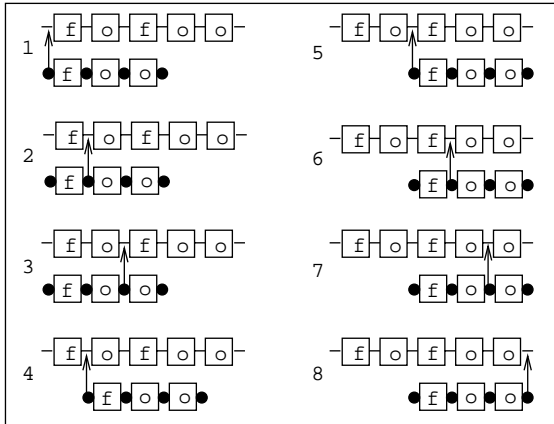
Using just these five constructs, we can create patterns of surprising utility and complexity. In fact, many of the other regex constructs are shorthands (and possibly optimizations) for patterns you can create with these simple constructs.

To get the most out of regular expressions, one requires an understanding of the kinds of things that go on during a pattern match. For the next few pages, we will construct simple patterns using the elements above to step through the processes of matching a pattern. The explanation here does not describe the underlying

machinery that actually performs the match but, instead, provides a description of how that machinery walks through a string to find a match.

To begin with, we will look at what happens during a match of  $m/f\circ f\circ\circ/$  against the string  $f\circ f\circ\circ$ . Although, in actuality, when a literal string is the pattern, an optimized search can be performed, here we will describe it here as a regular expression search.

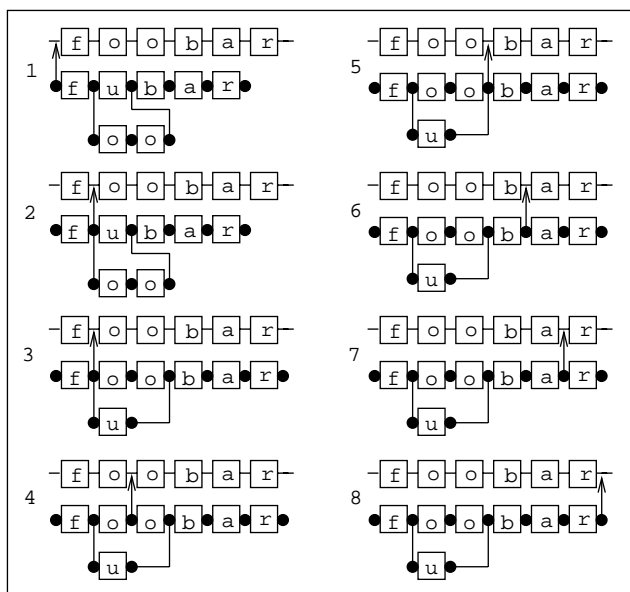
Figure 10.2 shows a graphical depiction of the steps required for our simple pattern. At each step, the target string is above the regular expression. At step 1, the regex is lined up below the beginning of the target string with a pointer representing the current position in both the regex and the target string. The first regex component is checked against the first position in the string. A match is found so the pointer moves ahead one position. The second position also matches the second regex component, and the pointer moves ahead again. Now the third regex component fails to match at the current position in the string. The pointer is returned to the beginning of the regex, and the whole regex is bumped ahead one position in the string (step 4), and the process repeats. The regex immediately fails and is bumped ahead again (step 5). Finally, the first component of the regex matches the current position so the pointer moves ahead again. The second and third components match in turn, and the pointer arrives at the end of the regex indicating a successful match.



**Figure 10.2** Stepping through a simple pattern match

This all seems relatively straightforward perhaps—and a long way to describe a simple match sequence that we could have performed in our head at a glance—but looking at these little steps will help us later when constructing more complex regular expressions.

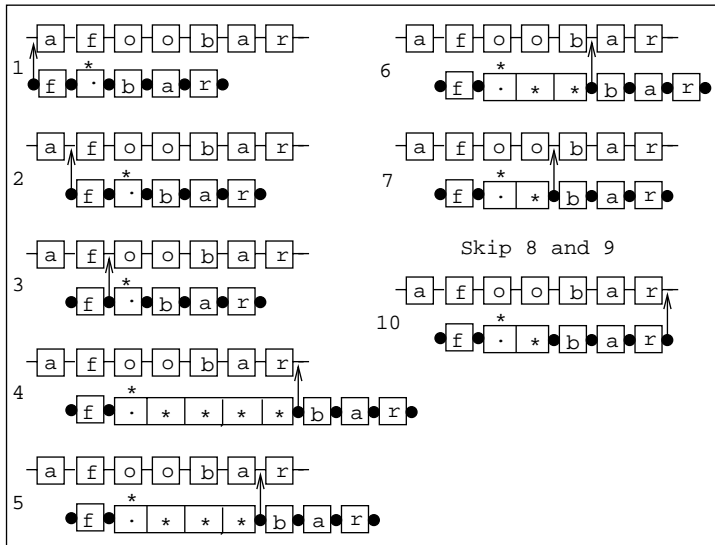
Let's consider a slightly more complex pattern: `m/f(u|oo)bar/` on the string `foobar`. We will only consider the grouping and alternation aspects here, not the capturing behavior. Again, we can easily see the match right away, but Perl will have to tackle the operation in simple steps once again. Figure 10.3 represents the steps taken in searching for this pattern. Here I have placed the second alternative below the first with connecting lines to show where it belongs in the regex sequence (if it is needed). The process begins as before. The first component matches at the first position in the target string, so the pointer moves ahead one position. Now the second component fails at the current position. The underlying search machinery sees that it has an alternative component to try here so it tries again with the second alternative in place. Now the two sub-components of the second regex element match in turn, as do the remaining components, and the pointer progresses to the end of the regex indicating success again.



**Figure 10.3 Stepping through a pattern match with alternation**

Finally, let's consider a simple case that utilizes the star (repetition) and the dot (wildcard) elements. Here we will use a pattern of `m/f.*bar/` against the target string `afoobar`. Figure 10.4 represents the steps taken in this matching process.

Here we see the failure at step 1. The regex bumps ahead to position 2 in the string. The first component matches here so the pointer moves ahead. Now, the dot can match anything, and the star says to match zero-or-more, so this component stretches right to the end of the string (step 4) leaving markers (i.e., the stars



**Figure 10.4** Stepping through pattern match with iteration

inside the boxes within this component) at each position . Now, the regex realizes it cannot match the next regex component because it is at the end of the string. So it returns to the second component, .\*, backs up one marker, and tries the third component once again (step 5). This backstepping (backtracking) continues until step 7 when the third component does match. After this, the remaining components match in turn (steps 8 and 9, not shown) until we reach the end of the regex and have a successful overall match.

This might seem like an odd way for the star operator to work, grabbing as much as it can, then releasing one step at a time and testing the remaining components. This is what is commonly referred to as a “greedy” or “maximal” matching quantifier. In section 10.4, we will introduce a non-greedy equivalent that first tries taking zero positions in the string and tests if the rest of the regex matches. If that fails, it takes one position from the string and tests the remaining regex components and so on until the match either succeeds or fails overall.

Now that we have a sense of the process of matching, we can begin to introduce the remaining regular expression elements shown in figure 10.1.

## 10.2 *The character class*

Another common feature of Perl’s regex language is the character class. In simple terms, a character class is just an easy way of providing a list of characters from which to choose a match for a given position in the string. A character class con-

sists of a set of square brackets enclosing the list of character choices. For example, the pattern `m/f[uo]bar/` would match either `fubar` or `fobar`, but not `fuobar` or `fbar`. Think of this pattern as equivalent in meaning to `m/f(u|o)bar/`. In reality, a character class is optimized and generally much more efficient than an alternation sequence. So, while the above two patterns mean the same thing in terms of what gets matched, the character class version will run more quickly.

You can specify a range of characters within a character class by using a dash between two characters. The character class `[a-f]` will match one lower case letter from `a` to `f` inclusive. If you want to include a dash (as a character not an operator) within a character class, you should place it first or last in the class: `[a-z-]` matches one character that is either a lower case letter or a dash.

A useful variation of the character class is the negated character class, which matches one character not in the list of characters in the class. To create a negative character class, you use the `^` character as the first character within the class: `[^a-z]` will match one character that is not a lower case letter. If you place the `^` anywhere but at the beginning of the character class, it is not special and simply becomes one of the characters in the list that may be matched.

### *10.2.1 Search and replace: capitalize headings*

So far we've been using examples that make it easy to demonstrate a point but are somewhat removed from real world use. Here we will look an example that addresses more typical usage of regular expressions—searching and replacing text in a file.

Let's say we have written or been given a lengthy file containing the source for a lengthy HTML document. The author of this document only capitalized the first word in each heading, and the boss wants us to edit the file so that every word in a heading is capitalized. A typical heading appears as

```
<H1> A typical heading </H1>
```

Luckily, in this case, the headings are all short and every heading (and its beginning and end tags) are located on individual lines in the file (we will consider cases of matching across multiple lines later in this chapter). The headings used in the document come in three levels marked by tags of `<H1>`, `<H2>`, and `<H3>`, although the author sometimes used lowercase (`<h2>`) versions of the tags.

We want to find every line containing a heading and capitalize each word within the heading. Before we do this, we need to review some of the special escapes that can be used within a double-quoted string. These can also be used in the replacement portion of a substitution operation and will greatly simplify our task. (These are listed in table 10.1 and were presented previously in chapter 4's

table 4.1.) The sequence we want in this case is the `\u` sequence, which uppercases the following letter:

```
print "here \uis an ex\uample"; # prints: here Is an exAmple

my $string = 'foobar';
$string =~ s/(b)/\u$1/;
print $string;                # prints: fooBar
```

The other thing we need to briefly introduce is the `/g` modifier, which causes the match or substitution operator to repeatedly search for the pattern throughout the target string. In the case of the substitution operator, it finds every occurrence within the string that matches the given pattern, and it replaces the matching text with whatever is in the replacement portion of the operator:

```
$string = 'foobar baz blob';
$string =~ s/(b)/\u$1/g;
print $string;                # prints: fooBar Baz BloB
```

Now we can tackle our main problem, capitalizing every word within a heading in the document. A simple way to do this is to search for a line containing an opening heading tag, then find the first letter of each word. Any word will start right after a space character except the first word that might begin right after the opening tag as in `<H1>This is the heading</H1>`. We already know that the first word is capitalized so we only need worry about the rest. The expression that performs the task is simply

```
s/ ([a-z])/ \u$1/g
```

This says to find a space followed by a lowercase letter—which is captured into the special `$1` variable—and replace the matched text with a space, followed by the same letter in uppercase. Let's assume the document is in a file called *article.html*. We can simply rename this file as *article.html.bak* and use the following program to create a new version of *article.html*:

```
#!/usr/bin/perl -w
use strict;
open(INPUT, 'article.html.bak') || die "can't open file: $!";
open(OUTPUT, '>article.html')   || die "can't open file: $!";
while(<INPUT>) {
    if ( m/<[Hh][1-3]/ ) {
        s/ ([a-z])/ \u$1/g;
    }
    print OUTPUT $_;
}
```

A more general version of this program would simply read a file given on the command line and print the resulting output on standard output so that it could be redirected to another file:

```
#!/usr/bin/perl -w
use strict;
while( <> ) {
    if ( m/<[Hh][1-3]>/ ) {
        s/ ([a-z])/ \u$1/g;
    }
    print;
}
```

If you named this program `cap_heads`, you could run it from the command line like this:

```
perl cap_heads article.html.bak > article.html
```

This way you can use it to modify other such documents without having to edit the program to replace the filenames.

### 10.2.2 Character class shortcuts

There are three special escape sequences that stand for commonly used character classes. Each of these has a variant to stand for the corresponding negated character class. We saw each of these and some examples of their use in chapter 6. We repeat them here for review:

**Table 10.1** Escape sequences for commonly used character classes

Escape sequence	Description
\w	equivalent to: [a-zA-Z0-9_], a word character any letter or digit or an underscore character
\W	equivalent to: [^a-zA-Z0-9_], a non-word character any character that is not a letter, digit, or underscore
\d	equivalent to: [0-9], any single digit
\D	equivalent to: [^0-9], any non-digit character
\s	equivalent to: [\n\r\t], a whitespace character a space, newline, formfeed, return, or tab character
\S	equivalent to: [^\n\r\t], a non-whitespace character

## 10.3 Greedy quantifiers: take what you can get

Another greedy quantifier that operates similarly to the star is the plus (+) quantifier. This one matches one-or-more of the previous components. You can think of

`m/f(o+)bar/` as being the same as `m/f(oo*)bar/`, in that matching one-or-more is the same as matching one thing, then zero-or-more of the same thing. The pattern `m/fo*bar/` would match against the string `fbar`, matching an `f` then zero `o` characters followed by `bar`. The pattern `m/fo+bar/` would not match against `fbar` because there isn't at least one `o` following the `f` in that string.

The star is an indeterminate quantifier that can match any number of characters. The plus is only slightly determinate in that it must match at least one thing, but could match any number of additional characters. Perl also offers a few other greedy quantifiers with varying degrees of indeterminacy. These are listed in table 10.2 along with their meaning and an example with an equivalent formulation using constructs we already know:

**Table 10.2 Greedy quantifiers**

Quantifier	Description
<code>?</code>	match zero-or-one time
<code>m/fo?bar/</code>	equivalent to <code>m/f(o )bar/</code>
<code>{n}</code>	match exactly <code>n</code> times
<code>m/fo{2}bar/</code>	equivalent to <code>m/foobbar/</code>
<code>{min,}</code>	match min-or-more times
<code>m/fo{2,}bar/</code>	equivalent to <code>m/foo+bar/</code>
<code>{min, max}</code>	match at least <code>min</code> times, but at most <code>max</code> times
<code>m/fo{2,4}bar/</code>	equivalent to <code>m/f(oooooooloo)bar/</code>

In the first example above, the equivalent `m/f(o|)bar/` might seem strange. All the grouped alternation means is match an `o` or match nothing. Also, if the ordering of the alternatives in the last example surprised you, remember that these are all greedy quantifiers and will first try to match as much as possible (or as much as they are allowed) before trying lesser amounts.

## 10.4 Non-greedy quantifiers: take what you need

Often, greedy quantifiers are simply too greedy for your intended purpose. Consider trying to match and capture all the text on a line up to the first occurrence of `a15`:

```
$line = "one two three a15 four five six a15 seven\n";
$line =~ m/(.*)a15/;
print "$1\n"; # prints: one two three a15 four five six
```

What happened? The star is greedy and matched all the way to the end of the string and then backtracked until an `a15` could match. What we need is something

that will match as little as possible and then check the rest of the expression to see if it matches yet. All of the greedy quantifiers have a non-greedy form that is simply the quantifier followed by a question mark. A revised version of our example above now using a non-greedy star quantifier would be

```
$line = "one two three a15 four five six a15 seven\n";  
$line =~ m/(.*?)a15/;  
print "$1\n"; # prints: one two three
```

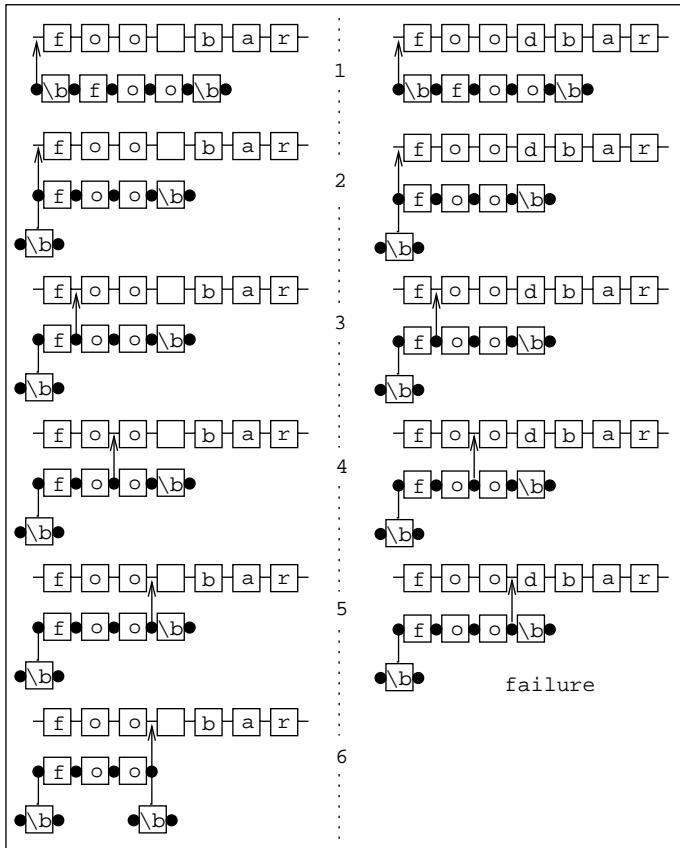
The `(.*?)` tries to match zero characters followed by an `a15`, then one character followed by an `a15`, and so on until it finally matches fourteen characters `one two three` and succeeds in finding a following `a15`. The other non-greedy versions of the quantifiers are given in figure 10.1 and operate in a similar manner.

## 10.5 *Simple anchors*

An anchor is a form of zero-width assertion. This means it matches not a character, but a position with certain properties. You have already seen two such elements in chapter 6: the `^` (caret) can be used to match at the beginning of a string, and the `$` can be used to match the end of a string. Another anchor type regex element that you've already seen is the word-boundary element `\b`. This matches at a position between a word character (`\w`) and a non-word character (`\W`) or between the beginning or end of a string and a `\w` character. To get an idea of what it means to match a position rather than a character, let's consider another simple example depicted graphically.

In figure 10.5 we step through running the pattern `m/\bfoo\b/` against the strings `foo bar` and `foodbar`. At step 1, `\b` matches at the start of the string (between the start of the string and the `f` character) in both cases. Because this is a zero-width assertion, the pointer remains pointing at the same position in the target string. In order to show the regex components in the place where they match on the string, we advance the pointer to the next regex component but drop the first component down below the regex. This is simply my way of showing that the `\b` and the `f` regex element both are successful at the first position in the string.

The pointer then advances along in the usual manner in each string until we hit the final `\b`. In the first case, there is again a match because the position between the `o` and the space is a word-boundary position. Thus the regex succeeds at step 6. In the second string, the position lies between an `o` and a `d`, which are both word characters, so the regex fails at this point. (Note: although not shown in the figure,



**Figure 10.5 Stepping through a pattern match with anchors**

the pointer would return to the beginning and the whole regex would repeatedly attempt to match the first component against every position in the string.)

The word-boundary anchor has a complement, the `\B` anchor, which matches at a position in a string between two word characters or two non-word characters.

When the `/m` modifier (we will discuss modifiers in the next chapter) is used on a match or substitution operator it means that the `^` and `$` anchors can match at the start and end of lines within a multi-line string. The `\A` and `\Z` anchors match only the beginning and end of a string respectively, regardless of whether it is a multi-line string or not.

The final simple anchor is the `\G` anchor, which works in conjunction with the `/g` modifier and anchors the match to the last position matched in a repeated match operation.

## 10.6 Grouping, capturing, and backreferences

Until now we have only used plain parentheses for grouping subexpressions. The disadvantage of this technique is that anything matched by the parenthesized sub-expression is captured and assigned to a special variable based on the position of the parentheses in the overall expression. This takes extra time and memory for the regex machinery, and, often, you are only interested in grouping a subpattern, not capturing its matching text. A form of parenthesization that will only group an expression is the `(?:subexpression)` form. For example, if you are not interested in capturing the matched text, the earlier example using `m/f(u|oo)bar/` would be better written as `m/f(?:u|oo)bar/`.

When capturing parentheses are used, the captured text is stored in a set of special variables (`$1`, `$2`, `$3` ...) where the digits reflect the order of occurrence of the subexpressions themselves counting from left to right. If the pattern `m/(foo(bar)baz) (xyz)/` did match against a target string, then `$1` would contain the string `foobarbaz`, `$2` would contain the string `bar`, and `$3` would contain the string `xyz`. These variables may be used within the replacement part of a substitution or in statements following a successful pattern match. These special variables are global, but automatically localized within their immediate enclosing block. They will retain their values until either another pattern match successfully matches or the current scope or block is exited. This is useful for extracting particular bits of data from within strings of text. (We saw examples of this sort of thing in chapter 6.)

Using capturing parentheses in a pattern also makes the captured text available later within the same pattern using the `\1`, `\2`, `\3` ... escape sequences. These are called backreferences because they refer back to previously matched text. A simplistic approach to searching a file for double words would be to use a pattern such as `m/\b(\w+)\s+\1\b/i`. This would match something resembling a word followed by one-or-more whitespace characters, followed by whatever word was matched by the capturing parentheses. The `/i` modifier tells the match operator to ignore case so we can catch doubled words that might differ in case. If used on multi-line strings—such as when the special `$/` variable is set to an empty string to read a file by paragraphs instead of line by line—we can still catch double words even if one is at the end of one line and the other is at the beginning of the next line. This is possible because we used the `\s` sequence instead of just a space. Remember that the `\s` sequence represents the character class `[ \n\f\r\t]` so the two words may be separated by one or more of any of those characters. Also note that we placed a word boundary following the backreference. This ensures that the backreferenced text here is not simply the beginning of a larger word such as in the perfectly logical string `This thistle is bristly`.

A program that makes use of this pattern to locate paragraphs containing doubled words and to highlight them somehow can be as simple as

```
#!/usr/bin/perl -w
use strict;
$/ = "; # read files in paragraph mode
while( <> ) {
    print if s/\b(\w+)(\s+)(\1)\b/*$1*$2*$3*/gi;
}
```

We captured the first occurrence of the word, the intervening whitespace, and the second occurrence of the word into three separate variables so we could replace the text with a few asterisks inserted to make the doubled words stand out. We also used the `/g` so that we could highlight multiple occurrences of doubled words within a paragraph. Jeffrey Friedl gives a more involved version of this program in his book<sup>1</sup> that allows an intervening tag such as an HTML tag between the doubled words, uses ANSI escapes to highlight the text, and prints out only highlighted lines rather than the whole paragraph. You are encouraged to add similar improvements to the above version of the program.

### 10.6.1 Prime number regex

Back in chapter 5, we developed a program to list all the prime numbers from 2 to N (where N was a number entered by the user of the program). That program was straightforward and relatively efficient. Here we will show another program to list prime numbers, one that is neither straightforward nor efficient, but nonetheless a marvelous example of something (of what we're just not sure).

If you visit <http://www.dejanews.com> and search the *comp.lang.perl.misc* archives for the terms "Abigail" and "prime," you'll eventually find a rather surprising usage of regular expressions to determine if a given number is prime. Abigail is a frequent poster to the *comp.lang.perl.misc* newsgroup and this example originated (as far as I know) as a clever little one-line program in one of Abigail's sign-off signatures. Since then, it has been commented on within the group on a few occasions and with a few variations, so you are sure to find quite a few articles when you search the archives. The following is an extended version that lists all the primes from 2 to N (where N is an argument to the program):

```
#!/usr/bin/perl -w
use strict;
my $N = shift @ARGV; # get the number
```

---

<sup>1</sup> Friedl, Jeffrey. *Mastering Regular Expressions*. Sebastopol, CA: O'Reilly and Associates, 1997.

```

for (my $number = 2; $number <= $N; $number++) {
    my $string = '1' x $number;
    print "$number is prime\n" if $string !~ m/^(11+)\1+$/;
}

```

Saving this program as *reprimes.pl* and running it from the command line produces the following results:

```

[danger:ajohnson:~]$ perl reprimes.pl 20
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime

```

But as I said earlier, this program is not very efficient—especially if you start trying larger values for *N*. For example, timing this program using a value of 1000 for *N* took 4.38 seconds compared to our chapter 5 “sieve” program, which took only 0.10 seconds to produce the same list.

Still this program does demonstrate backtracking and backreferences. In fact, all the real work is done using these mechanisms in the regular expression. Let’s take a closer look at what the main part of the program is doing. Each time through the loop, we have a new value of *\$number* to be tested for primality. We do this test in two steps:

```

my $string = '1' x $number;
print "$number is prime\n" if $string !~ m/^(11+)\1+$/;

```

First, we create a string of *\$number* characters (any character will do; we use the 1 character here) using the repetition operator (see chapter 4). So if *\$number* is 5, our string is 11111. Then we use the regex against the string to test if it contains a prime number of characters. Let’s look just at the regex itself, separated into its main components:

```

^          match the start of the string
(11+)     match 2 or more '1' characters and save in \1
\1+       try to match one or more of whatever we matched in \1
$          match the end of the string

```

The first thing to notice is that the pattern describes an entire string from beginning to end. The parentheses capture two or more characters, and the back-reference attempts to match one or more of the same text matched in the parentheses. If this match is successful, it means that the number of characters in the

string is evenly divisible by some integer greater than 1 and, hence, is not prime. So we use the negated form of the binding operator `!~` to say that the number is prime if it does not match the pattern.

## 10.7 Other anchors: lookahead and lookbehind

Perl's regex language has two additional, and very powerful, zero-width assertion mechanisms: lookahead assertions (negative and positive) and lookbehind assertions (negative and positive). The lookahead assertions have been available for quite some time, but the lookbehind assertions were introduced only in recent versions of Perl (5.005). The idea is similar to any other anchor—match a given position (not a character) in the string if the assertion is true.

Lookahead and lookbehind assertions come in two forms:

```
(?=subexpression)  positive lookahead
(?!subexpression)  negative lookahead

(?<=subexpression) positive lookbehind
(?<!subexpression) negative lookbehind
```

The positive lookahead assertion is true when the subexpression matches from the current position in the string, but, like the other anchors, it does not advance the pointer in the string—it merely tests to see if it would match. Similarly, a negative lookahead is successful when the subexpression does not match from the current position in the string.

```
m/ab(?=c)/;    #matches 'ab' only if followed by c
m/ab(?!c)/;    #matches 'ab' only if not followed by c
m/(?<=f)ab/;   #matches 'ab' only if preceded by an 'f'
m/(?<!f)ab/;   #matches 'ab' only if not preceded by an 'f'
```

Do not confuse the negative examples with a negative character class. If the target string is `f1ab`, then `m/ab[^c]/` will fail because `[^c]` must match one character (just not a `c`). The pattern `m/ab(?!c)/` will match against `f1ab`, however, because it contains an `ab` that is not followed by a `c`. Furthermore, a character class can only match one character from a list, but a lookahead can test any arbitrarily complex expression. Lookbehinds are not quite so powerful. The subexpression in a lookbehind subexpression must be a fixed-width lookbehind. It cannot contain any indeterminate quantifiers. The subexpression must describe a fixed-width substring.

### 10.7.1 Inserting commas in a number

One frequently asked question is how to insert commas into a number—for example, how to change “123456789” into “123,456,789.” This is not quite as easy as it may seem. The FAQs do provide a couple of answers. Before we look at the FAQ

solution that uses lookahead assertions, let's consider another example from Jeffrey Friedl's book. In this example, we also introduce the `/x` modifier, which allows us to write the regex pattern over multiple lines and insert comments. When the `/x` modifier is used with either a match or substitution operator, all whitespace within the pattern is ignored unless it is escaped with a backslash. Comments begin with the `#` character and continue to the end of the line:

```
$_ = 123456789;
s/          # begin substitution operator
  (\d{1,3}) # match and capture one-to-three digits
  (?=      # if they are followed by:
    (?:\d\d\d)+ # one-or-more groups of three digits
    (?!\d)      # that are not followed by a digit
  )          # end lookahead assertion
/$1,/gx;   # /g = perform substitution repeatedly
print;     #prints: 123,456,789
```

The `/x` modifier makes the above expression much more readable than the equivalent `s/(\d{1,3})(?=(?:\d\d\d)+(?!\d))/$1,/g`. This may require a little while to digest—take your time.

One limitation with the expression above is that the number cannot have a decimal portion: `123456.654321` would get transformed into `123,456.654,321`. If you want to insert commas correctly into decimal numbers, you first need to split the number at the decimal before you just commify the first portion and join the two sections again.

I came up with one of the FAQ answers to this problem shortly after reading Jeffrey's book when I needed to create reports out of large sets of simulation data. This version not only handles numbers with decimal portions; it can be used to commify all the numbers in a large block of text in one go. To achieve the results I wanted, I realized I would need something like a lookbehind assertion. At the time, Perl did not have them. Even now, Perl's lookbehind are fixed-width only, so they still will not solve the problem. The next best thing is to reverse the string itself and think about the problem backwards—a technique I call “Poor man's lookbehind.” This is simple to do because Perl has a built-in `reverse()` function that can be used to reverse a string:

```
$_ = '123456.789123 and 3.14159 and $15000.00';
$_ = reverse $_;
# now $_ contains: '00.00051$ dna 95141.3 dna 321987.654321'

s/
  (\d\d\d) # match 3 digits
  (?=\d)   # if they are followed by a digit
  (?!     # and they are not followed by
```

```

        \d*      # zero-or-more digits
        \.      # followed by a decimal point
    )         # end negative lookahead
/$1,/gx;
$_ = reverse $_;
print $_; # prints: 123,456.789123 and 3.14159 and $15,000.00

```

The way this works is fairly simple. First, let's consider an integer number such as 1234567. If we reverse this string, the number appears as 7654321. In this direction, we want to place commas after every three digits, but only if those three digits are followed by another digit. (How many doesn't matter, just as long as there is a least one additional digit.) A pattern of `s/(\d\d\d)(?=\d)/$1,/g` would first match 765 and then test for a following digit (which exists) and replace 765 with 765,. Because of the `/g` modifier, the regex would continue to try to match the pattern again and would find 432, which is also followed by a digit, and so another comma is inserted. There are no remaining groups of three digits, so the operation is completed, and reversing our number string provides us with the correctly formatted result.

Handling decimal numbers in a reversed string only requires one extra preventive measure. Given a number such as 1234.5678, the reversed form is 8765.4321. We know that any set of digits followed by zero-or-more digits and a decimal point is on the wrong side of the decimal point and should be ignored. All we need to add is another zero-width assertion—a negative lookahead in this case—to ensure that no set of digits on the left side of a decimal point will be matched. Our new pattern is `(\d\d\d)(?=\d)(?!\d*\.)`. Running this as the pattern in our substitution against the reversed number 8765.4321 would first match 876, test for a following digit (which succeeds), and then test again to see that that digit is not followed by zero-or-more digits and a decimal point. It is, and this assertions fails, so the regex moves ahead to try 765, which also fails the negative lookahead. The regex moves ahead until it can match three digits again with 432, which is followed by a digit and is not followed by zero-or-more digits and a decimal point. This succeeds, and a comma is correctly inserted.

The version in some earlier FAQs is in the form of a subroutine and has a slight error that will be noticed only when the subroutine is called in a list context, such as when it is used as an argument for the `print()` function. The correct subroutine version should be:

```

sub commify {
    my $input = shift @_;
    $input = reverse $input;
    $input =~ s/(\d\d\d)(?=\d)(?!\d*\.)><$1,>g;
    return scalar reverse $input;
}

```

## 10.8 Exercises

- 1 Will the pattern `m/(.*)(\d\d?)/` match the string `foo12bar`? If so, what will `$1` and `$2` contain? What would change if the pattern were `m/(.*)(\d\d)?bar/?`
- 2 Write a regex that will match a target string only if it contains only an integer number (all digits). Write one to match if the string is an integer or decimal number. Write one to match if the string is a positive or negative number including numbers in scientific format (i.e., `3e12`). See the *perlfaq4* page if you have difficulty.