Chapter 7. Regular Expressions

Perl has strong support for regular expressions, or *regexes* for short. This mini-language within Perl is a tight and powerful way to describe a family of strings that can match a pattern. It's one of the features that has made Perl so popular.

Today many languages have some sort of access to these powerful tools (perhaps as something called "Perl-Compatible Regular Expressions," or PCRE), but Perl is still out in front of most of them in its power and expressivity.

In the next three chapters, we will show you most of the regex features that you'll use in most programs. In this chapter, we'll show you the basics of the regular expression syntax. In Chapter 8, we show you the match operator and more sophisticated ways to employ patterns. Finally, in Chapter 9, we show you ways to use patterns to modify text.

Regexes will probably become one of your favorite parts of the language, at least for a while. But since regular expressions are tight and compact, they may also frustrate you to no end until you get used to them. That's normal. As you go through these chapters, try the examples as you read them. The more complex patterns build on what you've read earlier.

Sequences

Perl's regular expressions either match a string or they don't. There is no such thing as a partial match. And Perl doesn't look for a best match. Instead, it matches the leftmost, longest substring that satisfies the pattern.

NOTE

Some regular expression engines in other languages work differently, perhaps even looking for a "better" match *after* they find one that fits. See Jeffrey Friedl's *Mastering Regular Expressions* for the gory details.

The simplest sort of pattern is a *sequence*. You put literal characters next to each other to assert that you want to match a substring with those characters in that order. If you want to match the sequence `abba`, you put that between slashes:

```
$_ = "yabba dabba doo";
if (/abba/) {
  print "It matched!\n";
}
```

The forward slashes in the `if` expression are the match operator. It applies its pattern to the string in `$_`. The pattern is the part between the slashes. This might seem weird because this is the first time in this book that you've seen an operator that surrounds the value.

If the pattern matches the string in `$_`, the match operator returns true. Otherwise, it returns false. The match works by trying the pattern at the first position in the string. The first character in `$_` is a `y`, but the first character in the pattern is `a`. Those don't match, so Perl keeps looking.

The match operator then slides over one spot and tries the match at the next position, as shown in Figure 7-1. It matches the `a`—so far, so good. It tries to match the next character in the sequence, the `b`. It matches the first `b` in the string. Again, so far, so good. Then it matches the second `b`, and finally the last `a`. The match operator found the sequence in the string, so the pattern matches.

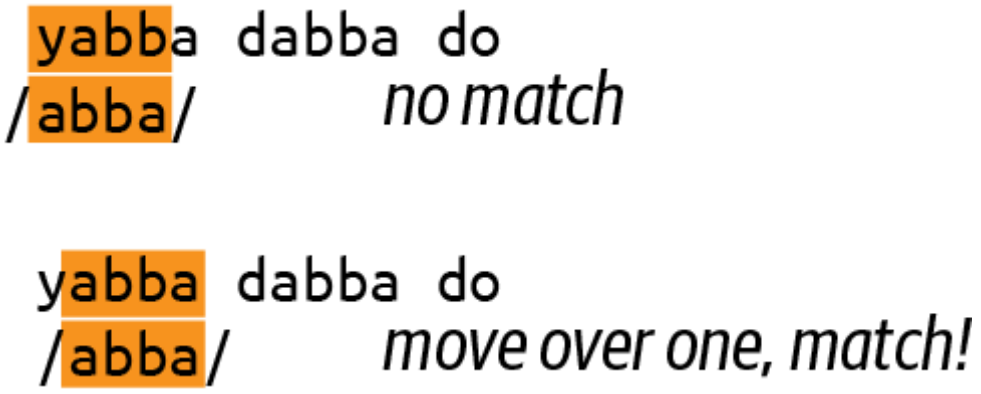

Figure 7-1. The pattern moves along the string looking for a match

Once the pattern is successful, the match operator returns true. This process matched the leftmost possible place to match. It doesn't need to find the second match within `dabba` because it already knows the string matches (although we'll show global matching in Chapter 8).

Inside a Perl pattern, whitespace is significant. Any whitespace you include in the pattern tries to match the same whitespace in `$_`. This example doesn't match because it looks for a space between the two `b`s:

```
$_ = "yabba dabba doo";
if (/ab ba/) {  # Won't match
  print "It matched!\n";
}
```

This pattern matches because the string in `$_` has a space between the `ba` and `da`:

```
$_ = "yabba dabba doo";
if (/ba da/) {  # Does match
  print "It matched!\n";
}
```

The pattern in the match operator is a double-quoted context; you can do the same sorts of things you can do in a double-quoted string. The special sequences such as `\t` and `\n` mean tab and newline just like they do in a double-quoted string. There are many ways to match a tab character:

```
/coke\tsprite/                          # \t for tab
/coke\N{CHARACTER TABULATION}sprite/    # \N{charname}
/coke\011sprite/                        # character number, octal
/coke\x09sprite/                        # character number, hex
/coke\x{9}sprite/                       # character number, hex
/coke${tab}sprite/                      # scalar variable
```

Perl first interpolates everything into the pattern and then compiles the pattern. If the pattern isn't a valid regex, you'll get an error. For example, a pattern with a single opening parenthesis is not valid (you'll see why later in this chapter):

```
$pattern = "(";
if (/$pattern/) {
  print "It matched!\n";
}
```

There's a pattern that matches every string. You can have the sequence of zero characters just like you can have the empty string:

```
$_ = "yabba dabba doo";
if (//) {
  print "It matched!\n";
}
```

If you follow the leftmost, longest rule, you'll work out that the empty pattern matches at the beginning of the string where it always finds the sequence of zero characters.

Practice Some Patterns

Now that you know the simplest form of regular expressions, you should try some yourself (especially if you've never used regular expressions). You'll learn more by trying them yourself than merely reading about them.

You know enough Perl to write a simple pattern tester now. You should replace the `PATTERN_GOES_HERE` with the pattern you'd like to test:

```
while( <STDIN> ) {
  chomp;
  if ( /PATTERN_GOES_HERE/ ) {
    print "\tMatches\n";
  }
  else {
    print "\tDoesn't match\n";
  }
}
```

Let's suppose that you want to test for `fred`. Change the pattern in the program:

```
while( <STDIN> ) {
  chomp;
  if ( /fred/ ) {
    print "\tMatches\n";
  }
  else {
    print "\tDoesn't match\n";
  }
}
```

When you run this program, it waits for input. For each line you enter, it tries the match and prints the result:

```
$ perl try_a_pattern
Capitalized Fred should not match
  Doesn't match
Lowercase fred should match
  Matches
Barney will not match
  Doesn't match
Neither will Frederick
  Doesn't match
But Alfred will
  Matches
```

NOTE

Some IDEs come with tools that help you build and test a regular expression. There are many online tools that do the same, such as regexr.com.

Note that the input line with a capitalized `F` does not match. We haven't shown you a way to make a pattern case-insensitive yet. Also note that the `fred` in `Alfred` matches even though it's in the middle of a larger word. Later you'll see how to fix that too.

You have to change the program every time you want to try a new pattern. That's an annoying way to program. Since you can interpolate a variable into a pattern, you can take the first argument from the command line as the pattern:

```
while( <STDIN> ) {
  chomp;
  if ( /$ARGV[0]/ ) { # May be hazardous for your health
    print "\tMatches\n";
  }
  else {
    print "\tDoesn't match\n";
  }
}
```

This is a slightly hazardous way to program since that argument could be anything and Perl has regex features to execute arbitrary code. For our purposes of testing simple regular expressions, we can take the hazard. Note that your shell may require you to quote the pattern since some of the regex characters may be special shell characters too:

```
$ perl try_a_pattern "fred"
This will match fred
  Matches
But not Barney
  Doesn't match
```

You can run it with a different pattern without changing the program:

```
$ perl try_a_pattern "barney"
This will match fred (not)
  Doesn't match
But it will match barney
  Matches
```

Again, this is slightly hazardous and we don't endorse this for production code. For this chapter, however, it does a decent job. As you go through this chapter, you may want to use this program to try new patterns as we introduce them. Try the examples as you run into them; your comfort with regexes comes with practice!

The Wildcard

The dot, `.`, matches any single character except a newline. It's the first regex *metacharacter* we show:

```
$_ = "yabba dabba doo";
if (/ab.a/) {
  print "It matched!\n";
}
```

That single exception, the newline, might seem weird. Perl considers the common case where you read a line of input and want to match against that string. In that case, the trailing newline is merely the line separator and not an interesting part of the string.

Inside the pattern, the dot is not a literal character. Sometimes you might miss this because the metacharacter also matches the literal version. This matches because the dot wildcard can match the `!` at the end of the string:

```
$_ = "yabba dabba doo!";
if (/doo./) {               # matches
```

```
  print "It matched!\n";
}

$_ = "yabba dabba doo\n";
if (/doo./) {               # doesn't match
  print "It matched!\n";
}
```

If you want to match an actual dot, you need to escape it with the backslash:

```
$_ = "yabba dabba doo.";
if (/doo\./) {              # matches
    print "It matched!\n";
}
```

The backslash is the second metacharacter (and we'll stop counting now). This means to match a literal backslash, you have to escape that too:

```
$_ = 'a real \\ backslash';
if (/\\/) {                 # matches
  print "It matched!\n";
}
```

Perl v5.12 added another way to write "any character except a newline." If you don't like the dot, you can use `\N`:

```
$_ = "yabba dabba doo!";
if (/doo\N/) {               # matches
  print "It matched!\n";
}

$_ = "yabba dabba doo\n";
if (/doo\N/) {               # doesn't match
  print "It matched!\n";
}
```

You'll see more about the `\N` in "Character Class Shortcuts".

Quantifiers

You can repeat parts of a pattern with a *quantifier*. These metacharacters apply to the part of the pattern that comes directly before them. Some people call these *repeat* or *repetition* operators.

The easiest quantifier is the question mark, `?`. It asserts that the preceding item shows up zero or one times (or, in human speak, that item is optional). Suppose that some people write `Bamm-bamm` while others write `Bammbamm` without the hyphen. You can match either by making the `-` optional:

```
$_ = 'Bamm-bamm';
if (/Bamm-?bamm/) {
  print "It matched!\n";
}
```

Try it with your test program using different ways to type Bamm-Bamm's name:

```
$ perl try_a_pattern "Bamm-?bamm"
Bamm-bamm
  Matches
Bammbamm
  Matches
Are you Bammbamm or Bamm-bamm?
  Matches
```

In that last line, which version of Bamm-Bamm's name matches? Perl starts at the left and shifts the pattern along the string until it matches. The first possible match is `Bammbamm`. Once Perl matches that, it stops, even though there is a longer possible match later in the string. Perl matches the leftmost substring; it doesn't even know about the later one because it didn't have to look that far ahead to know the string matched.

The next quantifier is the star, `*`. It asserts that the preceding item shows up zero or more times. That seems like a weird way to say it, but it means

it can be there or not. It's optional, but it can be there as many times as it likes:

```
$_ = 'Bamm-----bamm';
if (/Bamm-*bamm/) {
  print "It matched!\n";
}
```

Since you use the `*` after the hyphen, there can be any number of hyphens (including zero!) in the string. This is more handy for a variable amount of whitespace. Suppose you had the possibility of several spaces between the two parts of the name:

```
$_ = 'Bamm      bamm';
if (/Bamm *bamm/) {
  print "It matched!\n";
}
```

You could have another pattern to find a variable number of characters between the `B` and an `m`:

```
$_ = 'Bamm      bamm';
if (/B.*m/) {
  print "It matched!\n";
}
```

The longest part of the leftmost longest rule shows up here. The `.*` can match zero or more of any character except the newline, and so it does. In the process of matching, the `.*` matches the rest of the string all the way to the end of the string. We say that the quantifiers are *greedy* because they'll match as much as they can right away. Perl also has non-greedy matching, which you'll see in Chapter 9.

NOTE

In reality, *perl* employs various optimization tricks to make matching faster, so a greedy `.*` might be slightly less greedy if the match operator knows it can do a little less work.

But the next part of the pattern can't match, because it is already at the end of the string. Perl then starts backtracking (or unmatching) so that it can satisfy the rest of the pattern. It only needs to back up one character for the rest of the pattern (just the `m`) to match. Thus, the pattern matches from the first `B` to the very last `m` because that's the longest match it can make.

This also means that a pattern with a `.*` at the start or end does more work than it needs to do. Since the `.*` can match zero characters, these patterns don't need it:

```
$_ = 'Bamm      bamm';
if (/B.*/) {
  print "It matched!\n";
}

if (/.*B/) {
  print "It matched!\n";
}
```

The `.*` can always match zero characters, so these patterns might as well match a single `B`:

```
$_ = 'Bamm      bamm';
if (/B/) {
  print "It matched!\n";
}
```

NOTE

The `Regexp::Debugger` module animates the process of matching so that you can see what the regex engine is doing. We show you how to install modules in Chapter 11. We show more of this in the "Watch regexes with Regexp::Debugger" blog post.

Where the `*` matches zero or more times, the `+` quantifier matches one or more times. If there must be at least one space, use the `+`:

```
$_ = 'Bamm      bamm';
if (/Bamm +bamm/) {
  print "It matched!\n";
}
```

Those repetition operators match "or more" repetitions. What if you want to match an exact number? You can put that number in braces. Suppose you want to match exactly three `b`s. You can note that with `{3}`:

```
$_ = "yabbbba dabbba doo.";
if (/ab{3}a/) {
  print "It matched!\n";
}
```

That matches inside `dabbba` because that's where exactly three `b`s are. This is a handy way to avoid manually counting the characters yourself.

The situation is a bit different if the quantifier is at the end of the pattern:

```
$_ = "yabbbba dabbba doo.";
if (/ab{3}/) {
  print "It matched!\n";
}
```

Now the pattern can match inside `yabbbba` even though there are more than three `b`s. It doesn't limit the number of characters in the string, just the number it will match in the pattern.

If those aren't enough for you, there's the generalized quantifier where you get to choose the minimum and maximum times something can repeat. You put the minimum and maximum times to repeat in braces, such as `{2,3}`. Going back to an earlier example, what if you wanted to allow for two to three `b`s inside the `abba`? You would specify the minimum and maximum times:

```
$_ = "yabbbba dabbba doo.";
if (/ab{2,3}a/) {
  print "It matched!\n";
}
```

This pattern will first try to match the `abbb` in `yabbbba`, but after three `b`s there's another *b*. Perl has to move along to keep looking for a match, which it finds in `dabbba` since it has at least two `b`s and at most three `b`s. That's the leftmost longest match.

You can specify a minimum number of repetitions with no maximum; just leave out the maximum. This one matches in `yabbbba` because it has at least three `b`s and is the leftmost match:

```
$_ = "yabbbba dabbba doo.";
if (/ab{3,}a/) {
  print "It matched!\n";
}
```

Likewise, you can specify a maximum with no minimum by using `0` as the least number:

```
$_ = "yabbbba dabbba doo.";
if (/ab{0,5}a/) {
  print "It matched!\n";
}
```

Perl v5.34 removes the requirement to literally specify `0`, in the same way that you can leave off the maximum. Now you can write the no-minimum case like `{,n}`:

```
use v5.34;
$_ = "yabbbba dabbba doo.";
if (/ab{,5}a/) {  # will match
  print "It matched!\n";
}
```

Using `999` will match because there's a maximum of 999 `b`s but no minimum. Four or three `b`s satisfy that:

```
use v5.34;
$_ = "yabbbba dabbba doo.";
if (/ab{,999}a/) {  # will match
  print "It matched!\n";
}
```

NOTE

Perl v5.34 allows you to add spaces inside the meta-curlies in a double-quoted context (which a pattern is one sort of), so `{m,n}` can also be `{ m,n }`, `{m , n}`, and so on. This applies to quantifiers as well as things like `\x{}`, `\N{NAME}`, and others.

Now you have more metacharacters to escape if you want the literal versions: `?`, `*`, `+`, and `{`. Before v5.26 you could get away with some instances of unescaped literal `{`, but as Perl has expanded its regex features, the `{` has been pressed into service to mean more things. An unescaped literal `}` is fine because it doesn't start anything and doesn't confuse Perl.

NOTE

Perl v5.28 temporarily relaxed the requirement to escape a literal `{` to give people more time to fix legacy code. Perl v5.30 re-added that requirement.

You can rewrite all of the quantifiers in terms of the general one, as shown in Table 7-1.

Table 7-1. Regular expression quantifiers and their generalized forms

Number to match	Metacharacter	Generalized form
Optional	`?`	`{0,1}`
Zero or more	`*`	`{0,}`
One or more	`+`	`{1,}`
Minimum with no maximum		`{3,}`
Minimum with maximum		`{3,5}`
No minimum with maximum		`{0,5}` or `{,5}` (v5.34)
Exactly		`{3}`

Grouping in Patterns

You can use parentheses to group parts of a pattern. So, parentheses are also metacharacters.

Remember that a quantifier only applies to the immediately preceding item. The pattern `/fred+/` matches strings like `freddddddddd` because the quantifier only applies to the `d`. If you want to match repetitions of the entire `fred`, you can group with parentheses, as in `/(fred)+/`. The quantifier applies to the entire group, so it matches strings like `fredfredfred`, which is more likely to be what you wanted.

The parentheses also give you a way to reuse part of the string directly in the match. You can use *back references* to refer to text that you matched in the parentheses, called a *capture group*. You denote a back reference as

a backslash followed by a number, like `\1`, `\2`, and so on. The number denotes the capture group.

NOTE

You may also see "memories" or "capture buffers" in older documentation and earlier editions of this book, but the official name is "capture group."

When you use the parentheses around the dot, you match any nonnewline character. You can match again whichever character you matched in those parentheses by using the back reference `\1`:

```
$_ = "abba";
if (/(.)\1/) {  # matches 'bb'
  print "It matched same character next to itself!\n";
}
```

The `(.)\1` says that you have to match a character right next to itself. On the first try, the `(.)` matches an `a`, but when it looks at the back reference, which says the next thing it must match is `a`, that fails. Perl moves over one position in the string and starts the match over, using the `(.)` to match the next character, which is a `b`. The back reference `\1` now says that the next character in the pattern is `b`, which Perl can match.

The back reference doesn't have to be right next to the capture group. The next pattern matches any four nonnewline characters after a literal `y`, and you use the `\1` back reference to denote that you want to match the same four characters after the `d`:

```
$_ = "yabba dabba doo";
if (/y(....) d\1/) {
  print "It matched the same after y and d!\n";
}
```

You can use multiple groups of parentheses, and each group gets its own back reference. Suppose you want to match a nonnewline character in a

capture group, followed by another nonnewline character in a capture group. After those two groups, you use the back reference `\2` followed by the back reference `\1`. In effect, you're matching a palindrome such as `abba`:

```
$_ = "yabba dabba doo";
if (/y(.)(.)\2\1/) { # matches 'abba'
  print "It matched after the y!\n";
}
```

How do you know which group gets which number? Just count the order of the opening parentheses and ignore nesting:

```
$_ = "yabba dabba doo";
if (/y((.)(.)\3\2) d\1/) {
  print "It matched!\n";
}
```

You might be able to see this easier if you write out the regular expression to see the different parts (although this isn't a valid regular expression until we show you the `/x` flag in Chapter 8):

```
(        # first open parenthesis,  \1
    (.)  # second open parenthesis, \2
    (.)  # third open parenthesis,  \3
    \3
    \2
)
```

Perl 5.10 introduced a new way to denote back references. Instead of using the backslash and a number, you can use `\g{N}`, where `N` is the number of the back reference that you want to use.

Consider the problem where you want to use a back reference next to a part of the pattern that is a number. In this regular expression, you want to use `\1` to repeat the character you matched in the parentheses and follow that with the literal string `11`:

```
$_ = "aa11bb";
if (/(.)\111/) {
  print "It matched!\n";
}
```

Perl has to guess what you mean there. Is that the back reference `\1`, `\11`, or `\111`? Perl will create as many back references as it needs, so it assumes that you meant the octal escape `\111`. Perl only reserves `\1` through `\9` for back references. After that, it does a bit of guessing to determine if it's a back reference or an octal escape.

By using `\g{1}`, you disambiguate the back reference and the literal parts of the pattern:

```
use v5.10;

$_ = "aa11bb";
if (/(.)\g{1}11/) {
  print "It matched!\n";
}
```

You could leave the curly braces off the `\g{1}` and just use `\g1`, but in this case you need the braces. Instead of thinking about it, we recommend just using the braces all the time, at least until you're more sure of yourself.

You can also use negative numbers. Instead of specifying the absolute number of the capture group, you can specify a *relative back reference*. You can rewrite the last example to use `-1` as the number to do the same thing:

```
use v5.10;

$_ = "aa11bb";
if (/(.)\g{-1}11/) {
  print "It matched!\n";
}
```

If you add another capture group, you change the absolute numbering of all the back references. The relative back reference, however, just counts from its own position and refers to the group right before it no matter its absolute number, so it stays the same:

```
use v5.10;

$_ = "xaa11bb";
if (/(.)(.)\g{-1}11/) {
  print "It matched!\n";
}
```

Alternation

The vertical bar (|), often called “or” in this usage, means that either the left side may match, or the right side. If the part of the pattern on the left of the bar fails, the part on the right gets a chance to match:

```
foreach ( qw(fred betty barney dino) ) {
  if ( /fred|barney/ ) {
    print "$_ matched\n";
  }
}
```

This outputs two names. One matches the left alternative and one matches the right:

```
fred matched
barney matched
```

You can have more than one alternative:

```
foreach ( qw(fred betty barney dino) ) {
  if ( /fred|barney|betty/ ) {
    print "$_ matched\n";
```

```
  }
}
```

This outputs three names:

```
fred matched
betty matched
barney matched
```

The alternation divides the pattern into sides, which might not be what you want. Suppose you want to match one of the Flintstones, but you don't care if it's Fred or Wilma. You might try this:

```
$_ = "Fred Rubble";
if( /Fred|Wilma Flintstone/ ) {  # unexpectedly matches
  print "It matched!\n";
}
```

It unexpectedly matches! The left alternative was simply `Fred` while the other alternative was `Wilma Flintstone`. Since `Fred` appears in `$_`, it matches. If you want to limit the alternation, use parentheses to group it:

```
$_ = "Fred Rubble";
if( /(Fred|Wilma) Flintstone/ ) {  # doesn't match
  print "It matched!\n";
}
```

Perhaps your string has an annoying mix of tabs and spaces. Your alternation can have either: `( |\t)`. To get one or more of those characters, apply the `+` quantifier:

```
$_ = "fred  \t \t  barney"; # could be tabs, spaces, or both
if (/fred( |\t)+barney/) {
  print "It matched!\n";
}
```

Applying the quantifier to the alternation as a group (created with the parentheses) is different than applying the quantifier to each item inside the alternation:

```
$_ = "fred  \t \t  barney"; # could be tabs, spaces, or both
if (/fred( +|\t+)barney/) {  # all tabs or all spaces
  print "It matched!\n";
}
```

And note the difference without the parentheses. This pattern still matches even though there's no `barney`:

```
$_ = "fred  \t \t  wilma";
if (/fred |\tbarney/) {
  print "It matched!\n";
}
```

The pattern matches `fred` followed by a space, or it matches a tab followed by `barney`. It's everything on the left side or everything on the right side. If you want to limit the reach of the alternation, parentheses are the way to go.

Now consider a way to make your pattern case-insensitive. Maybe some people capitalize the second part of Bamm-Bamm's name, but some people don't. You can have an alternation that matches either case:

```
$_ = "Bamm-Bamm";
if (/Bamm-?(B|b)amm/) {
  print "The string has Bamm-Bamm\n";
}
```

Finally, consider an alternation to match any lowercase letter:

```
/(a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z)/
```

That's annoying, but we have a better way to do that. Keep reading!

Character Classes

Character classes are sets of characters that can match at a single location in the pattern. You put those characters in square braces, like `[abcwxyz]`. At that position in the pattern, it can match any one of those seven characters. It's a little like alternations but with individual characters.

For convenience, you can specify a range of characters with a hyphen (-), so that class may also be written as `[a-cw-z]`. That didn't save much typing, but it's more usual to make a character class like `[a-zA-Z]` to match any one letter out of that set of 52, or `[0-9]` to match a digit:

```
$_ = "The HAL-9000 requires authorization to continue.";
if (/HAL-[0-9]+/) {
  print "The string mentions some model of HAL computer.\n";
}
```

If you wanted to match a literal hyphen, you either escape it or put the hyphen at the beginning or end:

```
[-a]        # hyphen or an a
[a-]        # hyphen or an a
[a\-z]      # hyphen or an a or a z
[a-z]       # lowercase letters from a to z
```

Inside the character class, a dot is a literal dot:

```
[5.32]      # matches a literal dot or a 5, 3, or 2
```

You may use the same character shortcuts in any double-quotish string to define a character, so the class `[\000-\177]` matches any seven-bit ASCII character. Inside those square braces, the `\n` and `\t` are still newlines and tabs. Remember that these patterns have their own mini-language, so these rules only apply inside regular expressions and not to other parts of Perl.

Now you have your second way to make a match case-insensitive. You can specify a position that can have the uppercase or lowercase version of a letter:

```
$_ = "Bamm-Bamm";
if (/Bamm-?[Bb]amm/) {
  print "The string has Bamm-Bamm\n";
}
```

Sometimes it's easier to specify the characters you want to leave out rather than the ones within the character class. A caret (`^`) at the start of the character class creates the complement of it:

```
[^def]      # anything not d, e, or f
[^n-z]      # not a lowercase letter from n to z
[^n\-z]     # not an n, -, or z
```

This is handy when the list of characters that you don't want to match is shorter than the list of ones you do want.

Character Class Shortcuts

Some character classes are so common that Perl provides shortcuts, shown in Table 7-2. For example, you could rewrite the earlier example for spaces or tabs to accept other sorts of whitespace between the names. The `\s` is a shortcut meaning "whitespace" (although it doesn't match every Unicode whitespace character: see `\p{Space}` later):

```
$_ = "fred  \t \t  barney";
if (/fred\s+barney/) {  # whitespace
  print "It matched!\n";
}
```

That's not exactly what you had before, because whitespace includes more than just tabs or spaces. That might not matter to you. If you only wanted horizontal whitespace, you could use the `\h` shortcut introduced in v5.10:

```
$_ = "fred  \t \t  barney";
if (/fred\h+barney/) {  # any whitespace
  print "It matched!\n";
}
```

NOTE

In Perl versions before v5.18, the `\s` didn't match the vertical tab: see the blog post "Know your character classes under different semantics" for more information.

You can abbreviate the character class for any digit as `\d`. Thus, you could write the pattern from the example about HAL as `/HAL-\d+/` instead:

```
$_ = 'The HAL-9000 requires authorization to continue.';
if (/HAL-\d+/) {
  print "The string mentions some model of HAL computer.\n";
}
```

The shortcut `\w` is the so-called "word" character, although its idea of a word isn't like a normal word at all. The "word" was actually meant as an *identifier* character: the ones that you can use to name a Perl variable or subroutine.

The `\R` shortcut, introduced in Perl 5.10, matches any sort of linebreak, meaning that you don't have to think about which operating system you're using and what it thinks a linebreak is since `\R` will figure it out. This means you don't have to sweat the difference between `\r\n`, `\n`, and the other sorts of line endings that Unicode allows. It doesn't matter to you if there are DOS or Unix line endings. The `\R` isn't strictly a character class shortcut even though we note it here. It can match the two-character sequence `\r\n`, while real character classes match exactly one character.

You'll see more about these in Chapter 8. The story is a bit more complicated than you've seen here.

Negating the Shortcuts

Sometimes you may want the opposite of one of these three shortcuts. That is, you may want `[^\d]`, `[^\w]`, or `[^\s]`, meaning a nondigit character, a nonword character, or a nonwhitespace character, respectively. That's easy enough to accomplish by using their uppercase counterparts: `\D`, `\W`, and `\S`. These match any character that their counterpart would *not* match.

Table 7-2. ASCII character class shortcuts

Shortcut	Matches
\d	Decimal digits
\D	Not a decimal digit
\s	Whitespace
\S	Not whitespace
\h	Horizontal whitespace (v5.10 and later)
\H	Not horizontal whitespace (v5.10 and later)
\v	Vertical whitespace (v5.10 and later)
\V	Not vertical whitespace (v5.10 and later)
\R	Generalized line ending (v5.10 and later)
\w	“word”
\W	Not “word” characters
\n	Newline (not really a shortcut)
\N	Nonnewlines (stable in v5.18)

Any of these shortcuts will work either in place of a character class or inside the square brackets of a larger character class. For instance, `[\s\d]` will match whitespace and digits. Another compound character class is `[\d\D]`, which means any digit or any nondigit. That is to say, any character at all! This is a common way to match any character, including a newline.

Unicode Properties

Unicode characters know something about themselves; they aren't just sequences of bits. Every character knows not only what it is, but also what properties it has. Instead of matching on a particular character, you can match a type of character.

Each property has a name, which you can find in the perluniprops documentation. To match a particular property, you put the name in `\p{PROPERTY}`. For instance, some characters are whitespace, corresponding to the property name `Space`. To match any sort of space, you use `\p{Space}`:

```
if (/\p{Space}/) { # 25 different possible characters in v5.34
  print "The string has some whitespace.\n";
}
```

NOTE

The `\p{Space}` is slightly more expansive than `\s` because the property also matches NEXT LINE and NONBREAKING SPACE characters. It also matches LINE TABULATION (vertical tab), which `\s` did not match before v5.18.

If you want to match a digit, you use the `Digit` property, which matches the same characters as `\d`:

```
if (/\p{Digit}/) { # 650 different possible characters in v5.34
  print "The string has a digit.\n";
}
```

Those are both much more expansive than the sets of characters you may have run into. Some properties are more specific, though. How about matching two hex digits, `[0-9A-Fa-f]`, next to each other:

```
if (/\p{AHex}\p{AHex}/) { # 22 different possible characters
  print "The string has a pair of hex digits.\n";
}
```

You can also match characters that *don't* have a particular Unicode property. Instead of a lowercase `p`, you use an uppercase one to negate the property:

```
if (/\P{Space}/) { # Not space (many many characters!)
  print "The string has one or more nonwhitespace characters.\n";
}
```

Perl uses the properties named by the Unicode Consortium (with a few exceptions) and adds some of its own for convenience. They are listed in perluniprops.

Anchors

By default, if a pattern doesn't match at the start of the string, it can "float" down the string trying to match somewhere else. But there are a number of anchors that may be used to hold the pattern at a particular point in a string.

The `\A` anchor matches at the absolute beginning of a string, meaning that your pattern will not float down the string at all. This pattern looks for an `https` only at the start of the string:

```
if ( /\Ahttps?:/ ) {
  print "Found a URL\n";
}
```

The anchor is a *zero-width assertion*, meaning that it matches a condition at the current match position but doesn't match characters. In this case, the current match position has to be the beginning of the string. This keeps Perl from initially failing, moving over one character, and trying again.

If you want to anchor something to the end of the string, you use `\z`. This pattern matches `.png` only at the absolute end of the string:

```
if ( /\.png\z/ ) {
  print "Found a URL\n";
}
```

Why "absolute end of string"? We have to emphasize that nothing can come after the `\z`, because there is a bit of history here. There's another end-of-string anchor, the `\Z`, which allows an optional newline after it. That makes it easy to match something at the end of a single line without worrying about the trailing newline:

```
while (<STDIN>) {
  print if /\.png\Z/;
}
```

If you had to think about the newline, you'd have to remove it before the match and put it back on for the output:

```
while (<STDIN>) {
  chomp;
  print "$_\n" if /\.png\z/;
}
```

Sometimes you'll want to use both of these anchors to ensure that the pattern matches an entire string. A common example is `/\A\s*\Z/`, which matches a blank line. But this "blank" line may include some whitespace characters, like tabs and spaces, which are invisible to us. Any line that matches that pattern looks just like any other one on paper, so this pattern treats all blank lines as equivalent. Without the anchors, it would match nonblank lines as well.

The `\A`, `\Z`, and `\z` are Perl 5 regular expression features, but not everyone uses them. In Perl 4, where many people picked up their programming habits, the beginning-of-string anchor was the caret (`^`) and the end-of-string anchor was `$`. Those still work in Perl 5, but they morphed

into the beginning-of-line and end-of-line anchors, which are a bit different. We'll show more about those in Chapter 8.

Word Anchors

Anchors aren't just at the ends of the string. The word-boundary anchor `\b` matches at either end of a word. So you can use `/\bfred\b/` to match the word `fred` but not `frederick` or `alfred` or `manfred mann`. This is similar to the feature often called something like "match whole words only" in a word processor's search command.

Alas, these aren't words as we are likely to think of them; they're those `\w`-type words made up of ordinary letters, digits, and underscores. The `\b` anchor matches at the start or end of a group of `\w` characters. This is subject to the rules that `\w` is following, as we showed you earlier in this chapter.

In Figure 7-2, there's a line under each "word," and the arrows show the corresponding places where `\b` could match. There is always an even number of word boundaries in a given string, since there's an end-of-word for every start-of-word.

The "words" are sequences of letters, digits, and underscores; that is, a word in this sense is what's matched by `/\w+/`. There are five words in that sentence: `That`, `s`, `a`, `word`, and `boundary`. Notice that the quote marks around `word` don't change the word boundaries; these words are made of `\w` characters.

Each arrow points to the beginning or the end of one of the gray underlines, since the word-boundary anchor `\b` matches only at the beginning or the end of a group of word characters.

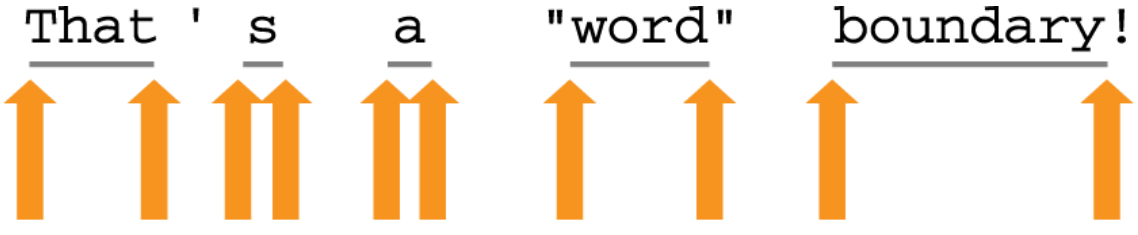

Figure 7-2. Word-boundary matches with `\b`

The word-boundary anchor is useful to ensure that you don't accidentally find `cat` in `delicatessen`, `dog` in `boondoggle`, or `fish` in `selfishness`. Sometimes you'll want just one word-boundary anchor, as when using `/\bhunt/` to match words like `hunt` or `hunting` or `hunter` but not `shunt`, or when using `/stone\b/` to match words like `sandstone` or `flintstone` but not `capstones`.

The nonword-boundary anchor is `\B`; it matches at any point where `\b` would not match. So the pattern `/\bsearch\B/` will match `searches`, `searching`, and `searched` but not `search` or `researching`.

Perl v5.22 and v5.24 added fancier anchors, but you need more matching skills to see how they work. You'll see them in Chapter 9 when we talk about the substitution operator.

Exercises

See "Answers to Chapter 7 Exercises" for answers to exercises.

Remember, it's normal to be surprised by some of the things that regular expressions do; that's one reason that the exercises in this chapter are even more important than the others. Expect the unexpected:

1. [10] Make a program that prints each line of its input that mentions `fred`. (It shouldn't do anything for other lines of input.) Does it match if your input string is `Fred`, `frederick`, or `Alfred`? Make a small text file with a few lines mentioning "fred flintstone" and his friends, then use that file as input to this program and the ones later in this section.
2. [6] Modify the previous program to allow `Fred` to match as well. Does it match now if your input string is `Fred`, `frederick`, or `Alfred`? (Add lines with these names to the text file.)
3. [6] Make a program that prints each line of its input that contains a period (`.`), ignoring other lines of input. Try it on the small text file from the previous exercise: does it notice `Mr. Slate`?
4. [8] Make a program that prints each line that has a word that is capitalized but not ALL capitalized. Does it match `Fred` but neither `fred`

nor `FRED`?

5. [8] Make a program that prints each line that has two of the same non-whitespace characters next to each other. It should match lines that contain words such as `Mississippi`, `Bamm-Bamm`, or `llama`.
6. [8] Extra-credit exercise: write a program that prints out any input line that mentions *both* `wilma` and `fred`.