Chapter 8. Matching with Regular Expressions

In Chapter 7, you visited the world of regular expressions. Now you'll see how that world fits into the world of Perl.

Matches with m//

You put patterns in pairs of forward slashes, like `/fred/`, but this is actually a shortcut for `m//`, the pattern match operator. As you saw with the `qw//` operator, you may choose any pair of delimiters to quote the contents. So you could write that same expression as `m(fred)`, `m<fred>`, `m{fred}`, or `m[fred]` using those paired delimiters, or as `m,fred,`, `m!fred!`, `m^fred^`, or many other ways using nonpaired delimiters.

NOTE

Nonpaired delimiters are the ones that don't have a different "left" and "right" variety; the same punctuation mark is used for both ends.

If you choose the forward slash as the delimiter, you may omit the initial `m`. Since Perl folks love to avoid typing extra characters, you'll see most pattern matches written using slashes, as in `/fred/`.

Of course, you should wisely choose a delimiter that doesn't appear in your pattern. If you wanted to make a pattern match the beginning of an ordinary web URL, you might start to write `/http:\/\//` to match the initial `"http://"`. But that would be easier to read, write, maintain, and debug if you used a better choice of delimiter: `m%http://%`. It's common to use curly braces as the delimiter. If you use a programmer's text editor, it probably has the ability to jump from an opening curly brace to the corresponding closing one, which can be handy in maintaining code.

If you're using paired delimiters, you shouldn't generally have to worry about using the delimiter inside the pattern, since that delimiter will usually be paired inside your pattern. That is, `m(fred(.*)barney)` and `m{\w{2,}}` and `m[wilma[\n \t]+betty]` are all fine, even though the pattern contains the quoting character, since each "left" has a corresponding "right." But the angle brackets (`<` and `>`) aren't regular expression metacharacters, so they may not be paired; if the pattern were `m{(\d+)\s*>=?\s*(\d+)}`, quoting it with angle brackets would mean having to backslash the greater-than sign so that it wouldn't prematurely end the pattern.

Match Modifiers

There are several modifier letters, sometimes called *flags*, which you can append as a group right after the ending delimiter. Some flags apply to the pattern and some change how the match operator behaves.

Case-Insensitive Matching with /i

To make a case-insensitive pattern match so that you can match `FRED` as easily as `fred` or `Fred`, use the `/i` modifier:

```
print "Would you like to play a game? ";
chomp($_ = <STDIN>);
if (/yes/i) {  # case-insensitive match
  print "In that case, I recommend that you go bowling.\n";
}
```

Matching Any Character with /s

By default, the dot (`.`) doesn't match a newline, and this makes sense for most "look within a single line" patterns. If you might have newlines in your strings, and you want the dot to be able to match them, the `/s` modifier will do the job. It changes every dot in the pattern to act like the character class `[\d\D]` does, which is to match any character, even if it is

a newline. Of course, you have to have a string with newlines for this to make a difference:

```
$_ = "I saw Barney\ndown at the bowling alley\nwith Fred\nlast night.\n";
if (/Barney.*Fred/s) {
  print "That string mentions Fred after Barney!\n";
}
```

Without the `/s` modifier, that match would fail, since the two names aren't on the same line.

This sometimes leaves you with a problem, though. The `/s` modifier applies to every `.` in the pattern. What if you wanted to still match any character except a newline? You could use the character class `[^\n]`, but that's a bit much to type, so v5.12 added the shortcut `\N` to mean the complement of `\n`.

If you don't like the `/s` modifier to make every `.` match any character, you can make your own character class to match any character. Just include each side of a character class shortcut pair: `[\D\d]` or `[\S\s]`, for example. The combination of every nondigit and digit should be everything.

Adding Whitespace with /x

The `/x` modifier makes most whitespace inside the pattern insignificant. That way, you can spread out the pattern to more easily see what's going on:

```
/-?[0-9]+\.?[0-9]*/     # what is this doing?
/ -? [0-9]+ \.? [0-9]* /x   # a little better
```

Since the `/x` allows whitespace inside the pattern, Perl ignores literal space or tab characters within the pattern. You could use a backslashed space or `\t` (among many other ways) to match these, but it's more common to use `\s` (or `\s*` or `\s+`) when you want to match whitespace. You

can also escape a literal space (although that's hard to show in text), or use `\x{20}` or `\040`.

Remember that Perl considers comments a type of whitespace, so you can put comments into that pattern to tell other people what you are trying to do:

```
/
  -?      # an optional minus sign
  [0-9]+  # one or more digits before the decimal point
  \.?     # an optional decimal point
  [0-9]*  # some optional digits after the decimal point
/x
```

Since the pound sign indicates the start of a comment, you need to use the escaped character, `\#`, or the character class, `[#]`, if you need to match a literal pound sign:

```
/
  [0-9]+  # one or more digits before the decimal point
  [#]     # literal pound sign
/x
```

Also, be careful not to include the closing delimiter inside the comments, or it will prematurely terminate the pattern. This pattern ends before you think it does:

```
/
  -?      # with / without -  <--- OOPS!
  [0-9]+  # one or more digits before the decimal point
  \.?     # an optional decimal point
  [0-9]*  # some optional digits after the decimal point
/x
```

When we started this section, we wrote "most whitespace" and then didn't tell you the rest of the story. You can't add insignificant whitespace inside a character class even with `/x`. Any characters inside the brackets,

including spaces and other whitespace, can match in the string. We're about to fix that, though.

Whitespace in character classes

Perl v5.26 added another way to add whitespace to a pattern. The `/xx` modifier does the same as `/x` but also allows you to add whitespace inside a character class without those whitespace characters becoming part of the class.

Consider this character class that matches six possible characters:

```
/ [abc123] /x  # matches a,b,c,1,2,3
```

That's not so hard to understand because it's short, but suppose those letters and numbers meant different things. There's a group of one sort of thing and there's a group of another sort of thing. You may want to separate those in the character class to show that, but it doesn't quite work because the space is now part of the character class:

```
/ [abc 123] /x  # matches a,b,c,1,2,3, or a space
```

To fix this, double up that `/x`:

```
use v5.26;
/ [abc 123] /xx  # matches a,b,c,1,2,3
/ [a-z 0-9] /xx  # matches lowercase letters or digits
```

Spread that over multiple lines if you like:

```
use v5.26;

/
   [
   abc
   123
```

```
   ]
/xx
```

But there's a limitation. While `/xx` lets you add that insignificant white-space, it doesn't let you add comments. The comment in this example is really part of the character class. This means you'll also match characters that you didn't expect:

```
use v5.26;

/
   [
   abc  # not a comment!
   123
   ]
/xx
```

Note that the `/xx` allows spaces in ranges. Put a space in there and it's still a range:

```
use v5.26;
/[0 - 9]/xx    # still 0-9
```

Combining Option Modifiers

If you want to use more than one modifier on the same match, just put them both at the end (their order isn't significant):

```
if (/barney.*fred/is) {  # both /i and /s
  print "That string mentions Fred after Barney!\n";
}
```

Or as a more expanded version with comments:

```
if (m{
  barney # the little guy
  .*     # anything in between
```

```
    fred   # the loud guy
  }six) {  # all three of /s and /i and /x
    print "That string mentions Fred after Barney!\n";
  }
```

Note the shift to curly braces here for the delimiters, allowing programmer-style editors to easily bounce from the beginning to the end of the regular expression.

Choosing a Character Interpretation

Perl v5.14 adds some modifiers that let you tell Perl how to interpret the characters in a match for two important topics: case folding and character class shortcuts. Everything in this section applies to v5.14 or later.

The `/a` tells Perl to use the ASCII interpretation of character classes, the `/u` tells Perl to use Unicode, and the `/l` tells Perl to respect the locale. Without these modifiers, Perl does what it thinks is the right thing based on the situations described in the perlre documentation. You use these modifiers to tell Perl exactly what you want despite whatever else is going on in the program:

```
use v5.14;

/\w+/a   # A-Z, a-z, 0-9, _
/\w+/u   # any Unicode word character
/\w+/l   # The ASCII version, and word chars from the locale,
         # perhaps characters like Œ from Latin-9
```

Which one is right for you? We can't tell you, because we don't know what you're trying to do. Each of them might be right for you at different times. Of course, you can always create your own character classes to get exactly what you want if the shortcuts don't work for you.

Now on to a harder issue. Consider case folding, where you need to know which lowercase letter you should get from an uppercase letter. This is part of the "Unicode bug" in Perl, where the internal representation de-

cides what answer you get. See the perlunicode documentation for the gory details.

If you want to match while ignoring case, Perl has to know how to produce lowercase characters. In ASCII, you know a *K*'s (0x4B) partner is a *k* (0x6B). In ASCII, you also know that a *k*'s uppercase partner is *K* (0x4B), which seems sensible but actually is not.

NOTE

You may like to peruse Unicode's case-folding rules. We'll show more in "Case Shifting".

In Unicode, things are not as simple, but it's still easy to deal with because the mapping is well defined. The Kelvin sign, *K* (U+212A), also has *k* (0x6B) as its lowercase partner. Even though *K* and *K* might look the same to you, they aren't to the computer. That is, lowercasing is not one-to-one. Once you get the lowercase *k*, you can't go back to its uppercase partner, because there is more than one uppercase character for it. Not only that, some characters, such as the ligature *ff* (U+FB00), have two characters as their lowercase equivalent—in this case, *ff*. The letter *ß* is *ss* in lowercase, but maybe you don't want to match that. A single `/a` modifier affects the character class shortcuts, but if you have two `/a` modifiers, it also tells Perl to use ASCII-only case folding:

```
/k/aai   # only matches the ASCII K or k, not Kelvin sign
/k/aia   # the /a's don't need to be next to each other
/ss/aai  # only matches ASCII ss, SS, sS, Ss, not ß
/ff/aai  # only matches ASCII ff, FF, fF, Ff, not ff
```

With locales it's not so simple. You have to know which locale you are using to know what a character is. If you have the ordinal value 0xBC, is that Latin-9's *Œ* or Latin-1's ¼ or something else in some other locale? You can't know how to lowercase it until you know what the locale thinks that value represents. We make the character with `chr()` to ensure that we get the right bit pattern regardless of the encoding issues:

```
$_ = <STDIN>;

my $OE = chr( 0xBC ); # get exactly what we intend

if (/$OE/i) {         # case-insensitive? Maybe not.
  print "Found $OE\n";
}
```

In this case, you might get different results depending on how Perl treats the string in `$_` and the string in the match operator. If your source code is in UTF-8 but your input is Latin-9, what happens? In Latin-9, the character *Œ* has ordinal value 0xBC and its lowercase partner *œ* has 0xBD. In Unicode, *Œ* is code point U+0152 and *œ* is code point U+0153. In Unicode, U+00BC is ¼ and doesn't have a lowercase version. If your input in `$_` is 0xBD and Perl treats that regular expression as UTF-8, you won't get the answer you expect. You can, however, add the `/l` modifier to force Perl to interpret the regular expression using the locale's rules:

```
use v5.14;

my $OE = chr( 0xBC ); # get exactly what we intend

$_ = <STDIN>;
if (/$OE/li) {    # that's better
  print "Found $OE\n";
}
```

If you always want to use Unicode semantics (which is the same as Latin-1) for this part, you can use the `/u` modifier:

```
use v5.14;

$_ = <STDIN>;
if (/Œ/ui) {   # now uses Unicode
  print "Found Œ\n";
}
```

If you think this is a big headache, you're right. No one likes this situation, but Perl does the best it can with the input and encodings it has to deal with. If only we could reset history and not make so many mistakes next time.

Beginning- and End-of-Line Anchors

What's the difference between the beginning-of-line and beginning-of-string? It comes down to the difference between how you think about lines and how the computer thinks about lines. When you match against the string in `$_`, Perl doesn't care what's in it. To Perl, it's just one big string, even if it looks like multiple lines to you because you have new-lines in the string. Lines matter to humans because we spatially separate parts of the string:

```
$_ = 'This is a wilma line
barney is on another line
but this ends in fred
and a final dino line';
```

Suppose your task, however, is to find strings that have `fred` at the end of any line instead of just at the end of the entire string. In Perl 5, you can do that with the `$` anchor and the `/m` modifier to turn on multiline matching. This pattern matches because in the multiline string, `fred` is at the end of a line:

```
/fred$/m
```

The addition of the `/m` changes how the old Perl 4 anchor works. Now it matches `fred` anywhere as long as it's either followed by a newline anywhere in the string or is at the absolute end of the string.

The `/m` does the same for the `^` anchor, which then matches either at the absolute beginning of the string or anywhere after a newline. This pattern matches because in the multiline string, `barney` is at the beginning of a line:

```
/^barney/m
```

Without the `/m`, the `^` and `$` act just like `\A` and `\Z`. However, since someone might come along later and add a `/m` switch, changing your anchors to something you didn't intend, it's safer to use only the anchors that mean exactly what you want and nothing more. But as we said, many programmers have habits they carried over from Perl 4, so you'll still see many `^` and `$` anchors that really should be `\A` and `\Z`. For the rest of the book, we'll use `\A` and `\Z` unless we specifically want multiline matching.

NOTE

The `re` module has a flags mode that allows you to set default flags for all match operators within its scope. Someone might make the `/m` flag the default.

Other Options

There are many other modifiers available. We'll cover those as we get to them, or you can read about them in the perlop documentation and in the descriptions of `m//` and the other regular expression operators that you'll see later in this chapter.

The Binding Operator =~

Matching against `$_` is merely the default; the *binding operator* (`=~`) tells Perl to match the pattern on the right against the string on the left, instead of matching against `$_`. For example:

```
my $some_other = "I dream of betty rubble.";
if ($some_other =~ /\brub/) {
  print "Aye, there's the rub.\n";
}
```

The first time you see it, the binding operator looks like some kind of assignment operator. But it's no such thing! It is simply saying, "This pattern match that would attach to `$_` by default—make it work with this string on the left instead." If there's no binding operator, the expression uses `$_` by default.

In the (somewhat unusual) next example, `$likes_perl` is set to a Boolean value according to what the user typed at the prompt. This is a little on the quick-and-dirty side because you discard the line of input itself. This code reads the line of input, tests that string against the pattern, then discards the line of input. It doesn't use or change `$_` at all:

```
print "Do you like Perl? ";
my $likes_perl = (<STDIN> =~ /\byes\b/i);
...  # Time passes...if ($likes_perl) {
  print "You said earlier that you like Perl, so...\n";
  ...
}
```

NOTE

Remember, Perl doesn't automatically store the line of input in `$_` unless the line-input operator (`<STDIN>`) is all alone in the conditional expression of a `while` loop.

Because the binding operator has fairly high precedence, the parentheses around the pattern test expression aren't required, so the following line does the same thing as the previous one—it stores the result of the test (and not the line of input) in the variable:

```
my $likes_perl = <STDIN> =~ /\byes\b/i;
```

The Match Variables

Parentheses normally trigger the regular expression engine's capturing features. The capture groups hold the part of the string matched by the part of the pattern inside parentheses. If there is more than one pair of parentheses, there will be more than one capture group. Each regular expression capture holds part of the original *string*, not part of the pattern. You could refer to these groups in your pattern using back references, but these groups also stick around after the match as the capture variables.

Since these variables hold strings, they are scalar variables; in Perl, they have names like `$1` and `$2`. There are as many of these variables as there are pairs of capturing parentheses in the pattern. As you'd expect, `$4` means the string matched by the fourth set of parentheses. This is the same string that the back reference `\4` would refer to during the pattern match. But these aren't two different names for the same thing; `\4` refers back to the capture within the pattern while it is trying to match, while `$4` refers to the capture of an already *completed* pattern match. For more information on back references, see the perlre documentation.

These match variables are a big part of the power of regular expressions because they let you pull out the parts of a string:

```
$_ = "Hello there, neighbor";
if (/\s([a-zA-Z]+),/) {    # capture the word between space and comma
  print "the word was $1\n";  # the word was there
}
```

Or you could use more than one capture at once:

```
$_ = "Hello there, neighbor";
if (/(\S+) (\S+), (\S+)/) {
  print "words were $1 $2 $3\n";
}
```

That tells you that the `words were Hello there neighbor`. Notice that there's no comma in the output. Because the comma is outside the capture parentheses in the pattern, there is no comma in capture two. Using

this technique, you can choose exactly what you want in the captures as well as what you want to leave out.

You could even have an empty match variable, if that part of the pattern might be empty. That is, a match variable may contain the empty string:

```
my $dino = "I fear that I'll be extinct after 1000 years.";
if ($dino =~ /([0-9]*) years/) {
  print "That said '$1' years.\n";  # 1000
}

$dino = "I fear that I'll be extinct after a few million years.";
if ($dino =~ /([0-9]*) years/) {
  print "That said '$1' years.\n";  # empty string
}
```

Remember that an empty string is different than an undefined one. If you have three or fewer sets of parentheses in the pattern, `$4` will be `undef`.

The Persistence of Captures

These capture variables generally stay around until the next *successful* pattern match. That is, an unsuccessful match leaves the previous capture values intact, but a successful one resets them all. This correctly implies that you shouldn't use these match variables unless the match succeeded; otherwise, you could be seeing a capture from some previous pattern. The following (bad) example is supposed to print a word matched from `$wilma`. But if the match fails, it's using whatever leftover string happens to be found in `$1`:

```
my $wilma = '123';
$wilma =~ /([0-9]+)/;   # Succeeds, $1 is 123
$wilma =~ /([a-zA-Z]+)/;  # BAD! Untested match result
print "Wilma's word was $1... or was it?\n";  # Still 123!
```

This is another reason a pattern match is almost always found in the conditional expression of an `if` or `while`:

```
if ($wilma =~ /([a-zA-Z]+)/) {
  print "Wilma's word was $1.\n";
} else {
  print "Wilma doesn't have a word.\n";
}
```

Since these captures don't stay around forever, you shouldn't use a match variable like `$1` more than a few lines after its pattern match. If your maintenance programmer adds a new regular expression between your regular expression and your use of `$1`, you'll be getting the value of `$1` for the second match rather than the first. For this reason, if you need a capture for more than a few lines, it's generally best to copy it into an ordinary variable. Doing this helps make the code more readable at the same time:

```
if ($wilma =~ /([a-zA-Z]+)/) {
  my $wilma_word = $1;
  ...
}
```

Later, in Chapter 9, you'll see how to get the capture value *directly* into the variable at the same time as the pattern match happens, without having to use `$1` explicitly.

Captures in Alternations

Captures may show up in alternations, and the same rules apply to their numbering: count the order of the opening parentheses. However, only one of the branches can match. In the two captures here, which one has a value?

```
if ( $name =~ /(F\w+)|(P\w+)/ ) { # Fred or Pebbles?
  print "1: $1\n2: $2\n";
}
```

With warnings on, one of those capture variables will complain about an uninitialized value.

To complicate it further, add a third capture outside the alternation, making that capture `$4`. Everything still works the same as what you already know:

```
/
(                 # $1
  (F\w+) |        # $2
  (P\d+)          # $3
)
\s+
  (\w+)           # $4
/x
```

What happens when the alternation gets another branch? That new branch becomes `$4` and the last capture moves up to `$5`:

```
/
(                 # $1
  (F\w+) |        # $2
  (P\d+) |        # $3
  (Dino)          # $4, new capture
)
\s+
  (\w+)           # $5 now
/x
```

That's no good. What you probably want is to count that grouped alternation as a single thing, and whichever capture matches gets the same number. That way, you know the capture number (there's only one) and new branches don't disturb the rest of the pattern.

Perl v5.10 added the branch reset operator, `(?|...)`, to handle that:

```
/
(?|               # anything in here is $1
  (F\w+) |
  (P\d+) |
  (Dino)
)
```

```
\s+
(\w+)           # $2
/x
```

In that contrived example, we can get the same thing by capturing the entire alternation because each branch captures everything:

```
/
(           # anything in here is $1
  F\w+ |
  P\d+ |
  Dino
)
\s+
(\w+)       # $2
/x
```

The branch reset is handy when some of the branches match extra text that they don't capture:

```
/
(?|             # anything in here is $1
  (Fr)ed   |
  (Peb)\d+ |
  (D)ino
)
\s+
(\w+)           # $2
/x
```

There's one more thing to note. Each branch can have a different number of captures, and the entire branch reset group will take up the same number of captures as the branch with the most. In this pattern, the third branch, `(D)(.)no`, has two captures, so the entire branch reset operator will have two captures. That's true even if the first branch, with only one capture, matches:

```
/
(?|             # takes up $1 and $2 always
  (Fr)ed   |
  (Peb)\d+ |
  (D)(.)no    # has two captures!
)
\s+
(\w+)           # $3
/x
```

Noncapturing Parentheses

So far you've seen parentheses that capture parts of a matched string and store them in the capture variables, but what if you just want to use the parentheses to group things? Consider a regular expression where we want to make part of it optional but only capture another part of it. In this example, you want "bronto" to be optional, but to make it optional you have to group that sequence of characters with parentheses. Later in the pattern, you use an alternation to get either "steak" or "burger," and you want to know which one you found:

```
if (/(bronto)?saurus (steak|burger)/) {
  print "Fred wants a $2\n";
}
```

Even if "bronto" is not there, its part of the pattern goes into `$1`. Perl just counts the order of the opening parentheses to decide what the capture variables will be. The part that you want to remember ends up in `$2`. In more complicated patterns, this situation can get quite confusing.

Fortunately, Perl's regular expressions have a way to use parentheses to group things but not trigger the capture groups, called *noncapturing parentheses*, and you write them with a special sequence. You add a question mark and a colon after the opening parenthesis, `(?:)`, and that tells Perl you only want to use these parentheses for grouping.

You change your regular expression to use noncapturing parentheses around “bronto,” and the part that you want to remember now shows up in `$1`:

```
if (/(?:bronto)?saurus (steak|burger)/) {
  print "Fred wants a $1\n";
}
```

Later, when you change your regular expression, perhaps to include a possible barbecue version of the brontosaurus burger, you can make the added “BBQ ” (with a space!) optional and noncapturing, so the part you want to remember still shows up in `$1`. Otherwise, you’d potentially have to shift all of your capture variable names every time you add grouping parentheses to your regular expression:

```
if (/(?:bronto)?saurus (?:BBQ )?(steak|burger)/) {
  print "Fred wants a $1\n";
}
```

Perl’s regular expressions have several other special parentheses sequences that do fancy and complicated things, like look-ahead, look-behind, embed comments, or even run code right in the middle of a pattern. You’ll have to check out the perlre documentation for the details.

If you want to do a lot of grouping but no capturing, you can use the `/n` flag added in v5.22. It turns all parentheses into noncapturing groups:

```
if (/(bronto)?saurus (BBQ )?(steak|burger)/n) {
  print "It matched\n"; # there is no $1 now
}
```

Named Captures

You can capture parts of the string with parentheses and then look in the number variables `$1`, `$2`, and so on to get the parts of the string that matched. Keeping track of those number variables and what should be in

them can be confusing even for simple patterns. Consider this regular expression that tries to match the two names in `$names`:

```
use v5.10;

my $names = 'Fred or Barney';
if ( $names =~ m/(\w+) and (\w+)/ ) { # won't match
  say "I saw $1 and $2";
}
```

You don't see the message from `say`, because the string has an `or` where you were expecting an `and`. Maybe you were supposed to have it both ways, so you change the regular expression to have an alternation to handle both `and` and `or`, adding another set of parentheses to group the alternation:

```
use v5.10;

my $names = 'Fred or Barney';
if ( $names =~ m/(\w+) (and|or) (\w+)/ ) { # matches now
  say "I saw $1 and $2";
}
```

Oops! You see a message this time, but it doesn't have the second name in it because you added another set of capture parentheses. The value in `$2` is from the alternation and the second name is now in `$3` (which we don't output):

```
I saw Fred and or
```

You could have used the noncapturing parentheses to get around this, but the real problem is that you have to remember which numbered parentheses belong to which data you are trying to capture. Imagine how much tougher this gets with many captures.

Instead of remembering numbers such as `$1`, v5.10 or later lets you name the captures directly in the regular expression. It saves the text it

matches in the hash named `%+` : the key is the label you used and the value is the part of the string that it matched. To label a match variable, you use `(?<LABEL>PATTERN)` where you replace `LABEL` with your own names. You label the first capture `name1` and the second one `name2` , and look in `$+{name1}` and `$+{name2}` to find their values:

```
use v5.10;

my $names = 'Fred or Barney';
if ( $names =~ m/(?<name1>\w+) (?:and|or) (?<name2>\w+)/ ) {
  say "I saw $+{name1} and $+{name2}";
}
```

Now you see the right message:

```
I saw Fred and Barney
```

Once you label your captures, you can move them around and add additional capture groups without disturbing the order of the captures:

```
use v5.10;

my $names = 'Fred or Barney';
if ( $names =~ m/((?<name2>\w+) (and|or) (?<name1>\w+))/ ) {
  say "I saw $+{name1} and $+{name2}";
}
```

Now that you have a way to label matches, you also need a way to refer to them for back references. Previously, you used either `\1` or `\g{1}` for this. With a labeled group, you can use the label in `\g{label}` :

```
use v5.10;

my $names = 'Fred Flintstone and Wilma Flintstone';

if ( $names =~ m/(?<last_name>\w+) and \w+ \g{last_name}/ ) {
  say "I saw $+{last_name}";
}
```

You can do the same thing with another syntax. Instead of using `\g{label}`, you use `\k<label>`:

```
use v5.10;

my $names = 'Fred Flintstone and Wilma Flintstone';

if ( $names =~ m/(?<last_name>\w+) and \w+ \k<last_name>/ ) {
  say "I saw $+{last_name}";
}
```

The `\k<label>` is essentially the same as `\g{label}`, but you can also use the `\g{}` syntax for a relative back reference such as `\g{N}`. In patterns that have two or more labeled groups with the same label, `\k<label>` and `\g{label}` always refer to the leftmost group.

Perl also lets you use the Python syntax. The sequence `(?P<LABEL>...)` forms a capture and `(?P=LABEL)` refers to that capture:

```
use v5.10;

my $names = 'Fred Flintstone and Wilma Flintstone';

if ( $names =~ m/(?P<last_name>\w+) and \w+ (?P=last_name)/ ) {
  say "I saw $+{last_name}";
}
```

The Automatic Match Variables

There are three more match variables that you get for free, whether the pattern has capture parentheses or not. That's the good news; the bad news is that these variables have weird names.

Now, Larry probably would have been happy enough to have called these by slightly less weird names, like perhaps `$gazoo` or `$ozmodiar`. But those are names that you just might want to use in your own code. To keep ordinary Perl programmers from having to memorize the names of

all of Perl's special variables before choosing their first variable names in their first programs, Larry has given strange names to many of Perl's built-in variables—names that "break the rules." In this case, the names are punctuation marks: `$&`, `$`` , and `$'`. They're strange, ugly, and weird, but those are their names. The part of the string that actually matched the pattern is automatically stored in `$&`:

```
if ("Hello there, neighbor" =~ /\s(\w+),/) {
  print "That actually matched '$&'.\n";
}
```

That tells you that the part that matched was `" there,"` (with a space, a word, and a comma). Capture one, in `$1`, has just the five-letter word `there`, but `$&` has the entire matched section.

Whatever came before the matched section is in `$``, and whatever was after it is in `$'`. Another way to say that is that `$`` holds whatever the regular expression engine had to skip over before it found the match, and `$'` has the remainder of the string that the pattern never got to. If you glue these three strings together in order, you'll always get back the original string:

```
if ("Hello there, neighbor" =~ /\s(\w+),/) {
  print "That was ($`)($&)($').\n";
}
```

The message shows the string as `(Hello)( there,)( neighbor)`, showing the three automatic match variables in action. We'll show more of those variables in a moment.

Any or all of these three automatic match variables may be empty, of course, just like the numbered capture variables. And they have the same scope as the numbered match variables. Generally, that means they'll stay around until the next successful pattern match.

Now, we said earlier that these three are "free." Well, freedom has its price. In this case, the price is that once you use any one of these auto-

matic match variables anywhere in your program, every regular expression will run a little more slowly.

Granted, this isn't a giant slowdown, but it's enough of a worry that many Perl programmers will simply never use these automatic match variables. Instead, they'll use a workaround. For example, if the only one you need is `$&`, just put parentheses around the whole pattern and use `$1` instead (you may need to renumber the pattern's captures, of course).

If you are using v5.10 or later, though, you can have your cake and eat it too. The `/p` modifier lets you have the same sort of variables while only suffering the penalty for that particular regular expression. Instead of `$``, `$&`, or `$'`, you use `${^PREMATCH}`, `${^MATCH}`, or `${^POSTMATCH}`. The previous examples then turn into:

```
use v5.10;
if ("Hello there, neighbor" =~ /\s(\w+),/p) {
  print "That actually matched '${^MATCH}'.\n";
}

if ("Hello there, neighbor" =~ /\s(\w+),/p) {
  print "That was (${^PREMATCH})(${^MATCH})(${^POSTMATCH}).\n";
}
```

Those variable names look a bit odd since they have the braces around the name and start with `^`. As Perl evolves, it runs out of names it can use for special names. Starting with a `^` means it won't clash with names that you might create (the `^` is an illegal character in a user-defined variable), but then it needs the braces to surround the entire variable name.

Match variables (both the automatic ones and the numbered ones) are most often used in substitutions, which you'll see in Chapter 9.

Precedence

With all of these metacharacters in regular expressions, you may feel that you can't keep track of the players without a scorecard. That's the prece-

dence chart, which shows us which parts of the pattern "stick together" the most tightly. Unlike the precedence chart for operators, the regular expression precedence chart is simple, with only five levels. As a bonus, this section will review all of the metacharacters that Perl uses in patterns. Table 8-1 shows the precedence, described here:

1. At the top of the precedence chart are the parentheses, `( )`, used for grouping and capturing. Anything in parentheses will "stick together" more tightly than anything else.
2. The second level is the quantifiers. These are the repeat operators—star (`*`), plus (`+`), and question mark (`?`)—as well as the quantifiers made with curly braces, like `{5,15}`, `{3,}`, `{,3}` (new in v5.34), and `{5}`. These always stick to the item they follow.
3. The third level of the precedence chart holds anchors and sequence. The anchors are the `\A`, `\Z`, `\z`, `^`, `$`, `\b`, and `\B` anchors you've already seen. There's a `\G` anchor that we don't cover in this book. Sequence (putting one item after another) is actually an operator, even though it doesn't use a metacharacter. That means that letters in a word will stick together just as tightly as the anchors stick to the letters.
4. The next-to-lowest level of precedence is the vertical bar (`|`) of alternation. Since this is near the bottom of the chart, it effectively cuts the pattern into pieces. It's here because you want the letters in the words in `/fred|barney/` to stick together more tightly than the alternation. If alternation were higher priority than sequence, that pattern would mean to match `fre`, followed by a choice of `d` or `b`, followed by `arney`. So, alternation is near the bottom of the chart, and the letters within the names stick together.
5. At the lowest level are the so-called *atoms* that make up the most basic pieces of the pattern. These are the individual characters, character classes, and back references.

Table 8-1. Regular expression precedence

Regular expression feature	Example
Parentheses (grouping or capturing)	(…), (?:…), (?<LABEL>…)
Quantifiers	a* a+ a? a{n,m}
Anchors and sequence	abc ^ $ \A \b \B \z \Z
Alternation	a|b|c
Atoms	a [abc] \d \1 \g{2}

Examples of Precedence

When you need to decipher a complex regular expression, you'll need to do as Perl does, and use the precedence chart to see what's really going on.

For example, `/\Afred|barney\z/` is probably not what the programmer intended. That's because the vertical bar of alternation is very low precedence; it cuts the pattern in two. That pattern matches either `fred` at the beginning of the string or `barney` at the end. It's much more likely that the programmer wanted `/\A(fred|barney)\z/`, which will match if the whole line has nothing but `fred` or nothing but `barney`. And what will `/(wilma|pebbles?)/` match? The question mark applies to the previous character, so that will match either `wilma` or `pebbles` or `pebble`, perhaps as part of a larger string (since there are no anchors).

The pattern `/\A(\w+)\s+(\w+)\z/` matches lines that have a "word," some required whitespace, and another "word," with nothing else before or after. That might be used to match lines like `fred flintstone`, for example. The parentheses around the words aren't needed for grouping, so they may be intended to save those substrings into the regular expression captures.

When you're trying to understand a complex pattern, it may be helpful to add parentheses to clarify the precedence. That's OK, but remember that grouping parentheses are also automatically capturing parentheses; use the noncapturing parentheses if you just want to group things.

And There's More

Although we've covered all of the regular expression features that most people are likely to need for everyday programming, there are still even more features. A few are covered in *Intermediate Perl*, but also check the perlre, perlrequick, and perlretut documentation for more information about what patterns in Perl can do.

A Pattern Test Program

When in the course of Perl events it becomes necessary for a programmer to write a regular expression, it may be difficult to tell just what the pattern will do. It's normal to find that a pattern matches more than you expected, or less. Or it may match earlier in the string than you expected, or later, or not at all.

This program is useful to test out a pattern on some strings and see just what it matches, and where:

```
while (<>) {           # take one input line at a time
  chomp;
  if (/YOUR_PATTERN_GOES_HERE/) {
    print "Matched: |$`<$&>$'|\n";  # the special match vars
  } else {
    print "No match: |$_|\n";
  }
}
```

NOTE

If you aren't using the ebook (from which you can cut and paste this code), you can get this from the Downloads section of the book's companion website.

This pattern test program is written for programmers to use, not end users; you can tell because it doesn't have any prompts or usage information. It will take any number of input lines and check each one against the pattern that you'll put in place of the string saying `YOUR_PATTERN_GOES_HERE`. For each line that matches, it uses the three special match variables (`$`` , `$&`, and `$'`) to make a picture of where the match happened. What you'll see is this: if the pattern is `/match/` and the input is `beforematchafter`, the output will say `|before<match>after|`, using angle brackets to show you just what part of the string was matched by your pattern. If your pattern matches something you didn't expect, you'll be able to see that right away.

Exercises

See "Answers to Chapter 8 Exercises" for answers to these exercises.

Several of these exercises ask you to use the test program from this chapter. You *could* manually type up this program, taking great care to get all of the odd punctuation marks correct, but you can also find it in the Downloads section of the book's companion website.

1. [8] Using the pattern test program, make a pattern to match the string `match`. Try the program with the input string `beforematchafter`. Does the output show the three parts of the match in the right order?
2. [7] Using the pattern test program, make a pattern that matches if any word (in the `\w` sense of word) ends with the letter `a`. Does it match `wilma` but not `barney`? Does it match `Mrs. Wilma Flintstone`? What about `wilma&fred`? Try it on the sample text file from the Exercises in Chapter 7 (and add these test strings if they weren't already in there).

3. [5] Modify the program from the previous exercise so that the word ending with the letter `a` is captured into `$1`. Update the code to display that variable's contents in single quotes, something like `$1 contains 'Wilma'`.
4. [5] Modify the program from the previous exercise to use named captures instead of relying on `$1`. Update the code to display that label name, something like `'word' contains 'Wilma'`.
5. [5] Extra-credit exercise: modify the program from the previous exercise so that immediately following the word ending in `a` it will also capture up to five characters (if there are that many characters, of course) in a separate capture variable. Update the code to display both capture variables. For example, if the input string says `I saw Wilma yesterday`, the up-to-five characters are "`  yest`" (with the leading space). If the input is `I, Wilma!`, the extra capture should have just one character. Does your pattern still match just plain `wilma`?
6. [5] Write a new program (*not* the test program!) that prints out any input line ending with whitespace (other than just a newline). Put a marker character at the end of the output line so as to make the whitespace visible.