7

Perl

Perl has been featured prominently in this book, and with good reason. It is popular, extremely rich with regular expressions, freely and readily obtainable, easily approachable by the beginner, and available for a remarkably wide variety of platforms, including pretty much all flavors of Windows, Unix, and the Mac.

Some of Perl's programming constructs superficially resemble those of C or other traditional programming languages, but the resemblance stops there. The way you wield Perl to solve a problem — *The Perl Way* — is different from traditional languages. The overall layout of a Perl program often uses traditional structured and object-oriented concepts, but data processing often relies heavily on regular expressions. In fact, I believe it is safe to say that **regular expressions play a key role in virtually all Perl programs**. This includes everything from huge 100,000-line systems, right down to simple one-liners, like

```
% perl -pi -e 's{([-+]?\d+(\.\d*)?)F\b}{sprintf "%.0fC",($1-32)*5/9}eg' *.txt
```

which goes through `*.txt` files and replaces Fahrenheit values with Celsius ones (reminiscent of the first example from Chapter 2).

In This Chapter

This chapter looks at everything regex about Perl,† including details of its regex flavor and the operators that put them to use. This chapter presents the regex-relevant details from the ground up, but I assume that you have at least a basic familiarity with Perl. (If you've read Chapter 2, you're already familiar enough to at least start using this chapter.) I'll often use, in passing, concepts that have not yet been examined in detail, and I won't dwell much on non-regex aspects of the language. It might be a good idea to keep the Perl documentation handy, or perhaps O'Reilly's *Programming Perl*.

† This book covers features of Perl as of Version 5.8.

Perhaps more important than your current knowledge of Perl is your *desire to understand more*. This chapter is not light reading by any measure. Because it's not my aim to teach Perl from scratch, I am afforded a luxury that general books about Perl do not have: I don't have to omit important details in favor of weaving one coherent story that progresses unbroken through the whole chapter. Some of the issues are complex, and the details thick; don't be worried if you can't take it all in at once. I recommend first reading the chapter through to get the overall picture, and returning in the future to use it as a reference as needed.

To help guide your way, here's a quick rundown of how this chapter is organized:

- "Perl's Regex Flavor" (☞ 286) looks at the rich set of metacharacters supported by Perl regular expressions, along with additional features afforded to raw regex literals.
- "Regex Related Perlisms" (☞ 293) looks at some aspects of Perl that are of particular interest when using regular expressions. *Dynamic scoping* and *expression context* are covered in detail, with a strong bent toward explaining their relationship with regular expressions.
- Regular expressions are not useful without a way to apply them, so the following sections provide all the details to Perl's sometimes magical regex controls:
  "The `qr/…/` Operator and Regex Objects" (☞ 303)
  "The Match Operator" (☞ 306)
  "The Substitution Operator" (☞ 318)
  "The Split Operator" (☞ 321)
- "Fun with Perl Enhancements" (☞ 326) goes over a few Perl-only enhancements to Perl's regular-expression repertoire, including the ability to execute arbitrary Perl code during the application of a regular expression.
- "Perl Efficiency Issues" (☞ 347) delves into an area close to every Perl programmer's heart. Perl uses a Traditional NFA match engine, so you can feel free to start using all the techniques from Chapter 6 right away. There are, of course, Perl-specific issues that can greatly affect in what way, and how quickly, Perl applies your regexes. We'll look at them here.

Perl in Earlier Chapters

Perl is touched on throughout most of this book:

- **Chapter 2** contains an introduction to Perl, with many regex examples.
- **Chapter 3** contains a section on Perl history (☞ 88), and touches on numerous regex-related issues that apply to Perl, such as character-encoding issues (including Unicode ☞ 105), match modes (☞ 109), and a long overview of metacharacters (☞ 112).

- **Chapter 4** is a key chapter that demystifies the Traditional NFA match engine found in Perl. Chapter 4 is extremely important to Perl users.
- **Chapter 5** contains many examples, discussed in the light of Chapter 4. Many of the examples are in Perl, but even those not presented in Perl apply to Perl.
- **Chapter 6** is an important chapter to the user of Perl interested in efficiency.

In the interest of clarity for those not familiar with Perl, I often simplified Perl examples in these earlier chapters, writing in as much of a self-documenting pseudo-code style as possible. In this chapter, I'll try to present examples in a more Perlish style of Perl.

Regular Expressions as a Language Component

An attractive feature of Perl is that regex support is so deftly built in as part of the language. Rather than providing stand-alone functions for applying regular expressions, Perl provides regular-expression *operators* that are meshed well with the rich set of other operators and constructs that make up the Perl language.

With as much regex-wielding power as Perl has, one might think that it's overflowing with different operators and such, but actually, Perl provides only four regex-related operators, and a small handful of related items, shown in Table 7-1.

Table 7-1: Overview of Perl's Regex-Related Items

Regex-Related Operators	**Modifiers**	**Modify How . . .**
`m/`*regex*`/`*mods* (☞ 306)	`/x` `/o`	regex is interpreted (☞ 292, 348)
`s/`*regex*`/`*replacement*`/`*mods* (☞ 318)	`/s` `/m` `/i`	engine considers target text (☞ 292)
`qr/`*regex*`/`*mods* (☞ 303)	`/g` `/c` `/e`	other (☞ 311, 315, 319)
`split(...)` (☞ 321)	**After-Match Variables** (☞ 299)	
Related Pragmas	`$1`, `$2`, etc.	captured text
`use charnames ':full';` (☞ 290)	`$^N` `$+`	latest/highest filled `$1`, `$2`, . . .
`use overload;` (☞ 341)	`@-` `@+`	arrays of indices into target
`use re 'eval';` (☞ 337)	`$`` `$&` `$'`	text before, of, and after match
`use re 'debug';` (☞ 361)	(best to avoid—see "Perl Efficiency Issues" ☞ 356)	
Related Functions	**Related Variables**	
`lc` `lcfirst` `uc` `ucfirst` (☞ 290)	`$_`	default search target (☞ 308)
`pos` (☞ 313) `quotemeta` (☞ 290)	`$^R`	embedded-code result (☞ 302)
`reset` (☞ 308) `study` (☞ 359)		

Perl is extremely powerful, but all that power in such a small set of operators can be a dual-edged sword.

Perl's Greatest Strength

The richness of variety and options among Perl's operators and functions is perhaps its greatest feature. They can change their behavior depending on the context in which they're used, often doing just what the author naturally intends in each differing situation. In fact, O'Reilly's *Programming Perl* goes so far as to boldly state "In general, Perl operators do exactly what you want...." The regex match operator `m/`*regex*`/`, for example, offers an amazing variety of different functionality depending upon where, how, and with which modifiers it is used.

Perl's Greatest Weakness

This concentrated richness in expressive power is also one of Perl's least-attractive features. There are innumerable special cases, conditions, and contexts that seem to change out from under you without warning when you make a subtle change in your code—you've just hit another special case you weren't aware of.† The *Programming Perl* quote in the previous paragraph continues "...unless you want consistency." Certainly, when it comes to computer science, there is a certain appreciation to boring, consistent, dependable interfaces. Perl's power can be a devastating weapon in the hands of a skilled user, but it sometimes seems with Perl, you become skilled by repeatedly shooting yourself in the foot.

Perl's Regex Flavor

Table 7-2 on the facing page summarizes Perl's regex flavor. It used to be that Perl had many metacharacters that no other system supported, but over the years, other systems have adopted many of Perl's innovations. These common features are covered by the overview in Chapter 3, but there are a few Perl-specific items discussed later in this chapter. (Table 7-2 has references to where each item is discussed.)

The following notes supplement the table:

- ⌈`\b`⌋ matches a backspace only in class; otherwise, it's a word boundary.
- Octal escapes accept two- and three-digit numbers.
- The ⌈`\x`*num*⌋ hex escape accepts two-digit numbers (and one-digit numbers, but with a warning if warnings are turned on). The ⌈`\x{`*num*`}`⌋ syntax accepts a hexadecimal number of any length.

† That they're innumerable doesn't stop this chapter from trying to cover them all!

Table 7-2: Overview of Perl's Regular-Expression Flavor

Character Shorthands		
☞ 114	(c)	**\a \b \e \f \n \r \t** *octal* **\x***hex* **\x{***hex***}** **\c***char*
Character Classes and Class-Like Constructs		
☞ 117		Classes: **[…] [^…]** (may contain POSIX-like **[:alpha:]** notation)
☞ 118		Any character except newline: *dot* (with **/s**, any character at all)
☞ 328		Forced match of single byte (can be dangerous): **\C**
☞ 125		Unicode combining sequence: **\X**
☞ 119	(c)	Class shorthands: **\w \d \s \W \D \S**
☞ 119	(c)	Unicode properties, scripts, and blocks: **\p{***Prop***}** **\P{***Prop***}**
Anchors and other Zero-Width Tests		
☞ 127		Start of line/string: **^** **\A**
☞ 127		End of line/string: **$ \z \Z**
☞ 315		End of previous match: **\G**
☞ 131		Word boundary: **\b \B**
☞ 132		Lookaround: **(?=…) (?!…) (?<=…) (?<!…)**
Comments and Mode Modifiers		
☞ 133		Mode modifiers: **(?***mods***-***mods***)** Modifiers allowed: **x s m i** (☞ 292)
☞ 134		Mode-modified spans: **(?***mods***-***mods***:…)**
☞ 134		Comments: **(?#…)** **#** ··· (from '#' until newline, or end of regex)
Grouping, Capturing, Conditional, and Control		
☞ 135		Capturing parentheses: **(…)** **\1 \2** ...
☞ 136		Grouping-only parentheses: **(?:…)**
☞ 137		Atomic grouping: **(?>…)**
☞ 138		Alternation: **|**
☞ 139		Greedy quantifiers: *** + ? {n} {n,} {x,y}**
☞ 140		Lazy quantifiers: ***? +? ?? {n}? {n,}? {x,y}?**
☞ 138		Conditional: **(?***if then* **|** *else***)** – "if" can be embedded code, lookaround, or **(***num***)**
☞ 327		Embedded code: **(?{…})**
☞ 327		Dynamic regex: **(??{…})**
In Regex Literals Only		
☞ 289	(c)	Variable interpolation: **$***name* **@***name*
☞ 290	(c)	Fold next character's case: **\l \u**
☞ 290	(c)	Case-folding span: **\U \L** ... **\E**
☞ 290	(c)	Literal-text span: **\Q** ... **\E**
☞ 290	(c)	Named Unicode character: **\N{***name***}** – optional; see page 290

(c) – may be used within a character class

- Perl's Unicode support is for Unicode Version 3.2.
- *dot* treats Unicode combining characters as separate characters (☞ 107). Also see ⌈\X⌋ (☞ 125).
- ⌈\w⌋, ⌈\d⌋, and ⌈\s⌋ fully support Unicode.
- Perl's ⌈\s⌋ does not match an ASCII vertical tab character (☞ 114).
- Unicode Scripts are supported. Script and property names may have the '`Is`' prefix, but they don't require it (☞ 123). Block names may have the '`In`' prefix, but require it only when a block name conflicts with a script name.

  The ⌈\p{L&}⌋ pseudo-property is supported, as well as ⌈\p{Any}⌋, ⌈\p{All}⌋, ⌈\p{Assigned}⌋, and ⌈\p{Unassigned}⌋.

  The long property names like ⌈\p{Letter}⌋ are supported. Names may have a space, underscore, or nothing between the word parts of a name (for example ⌈\p{Lowercase_Letter}⌋ may also be written as ⌈\p{LowercaseLetter}⌋ or ⌈\p{Lowercase Letter}⌋.) For consistency, I recommend using the long names as shown in the table on page 121.
- ⌈\p{^···}⌋ is the same as ⌈\P{···}⌋.
- Word boundaries fully support Unicode.
- Lookaround may have capturing parentheses.
- Lookbehind is limited to subexpressions that always match fixed-width text.
- The `/x` modifier recognizes only ASCII whitespace. The `/m` modifier affects only newlines, and not the full list of Unicode line terminators.

Not all metacharacters are created equal. Some "regex metacharacters" are not even supported by the regex engine, but by the preprocessing Perl gives to regex literals.

Regex Operands and Regex Literals

The final items in Table 7-2 are marked "regex literals only." A *regex literal* is the "*regex*" part of **`m/`*regex*`/`**, and while casual conversation refers to that as "the regular expression," the part between the '`/`' delimiters is actually parsed using its own unique rules. In Perl jargon, a regex literal is treated as a "regex-aware double-quoted string," and it's the result of that processing that's passed to the regex engine. This regex-literal processing offers special functionality in building the regular expression.

For example, a regex literal offers *variable interpolation*. If the variable `$num` contains `20`, the code `m/:.{$num}:/` produces the regex ⌈:.{20}:⌋. This way, you can

build regular expressions on the fly. Another service given to regex literals is automatic case folding, as with `\U`⋯`\E` to ensure letters are uppercased. As a silly example, `m/abc\Uxyz\E/` creates the regex ⌈`abcXYZ`⌋. This example is silly because if someone wanted ⌈`abcXYZ`⌋ they could just type `m/abcXYZ/` directly, but its value becomes apparent when combined with variable interpolation: if the variable `$tag` contains the string "`title`", the code `m{</\U$tag\E>}` produces ⌈`</TITLE>`⌋.

What's the opposite of a regex literal? You can also use a string (or any expression) as a regex operand. For example:

```
$MatchField = "^Subject:"; # Normal string assignment
   ⋮
if ($text =~ $MatchField) {
     ⋮
```

When `$MatchField` is used as an operand of **`=~`**, its contents are interpreted as a regular expression. That "interpretation" is as a plain vanilla regex, so variable interpolation and things like `\Q`⋯`\E` are not supported as they would be for a regex literal.

Here's something interesting: if you replace

```
$text =~ $MatchField
```

with

```
$text =~ m/$MatchField/
```

the result is exactly the same. In this case, there's a regex literal, but it's composed of just one thing—the interpolation of the variable `$MatchField`. The *contents* of a variable interpolated by a regex literal are *not* treated as a regex literal, and so things like **`\U`**⋯**`\E`** and $*var* within the value interpolated are *not* recognized. (Details on exactly how regex literals are processed are covered on page 292.)

If used more than once during the execution of a program, there are important efficiency issues with regex operands that are raw strings, or that use variable interpolation. These are discussed starting on page 348.

Features supported by regex literals

The following features are offered by regex literals:

- **Variable Interpolation** Variable references beginning with **`$`** and **`@`** are interpolated into the value to use for the regex. Those beginning with `$` insert a simple scalar value. Those beginning with `@` insert an array or array slice into the value, with elements separated by spaces (actually, by the contents of the **`$"`** variable, which defaults to a space).

  In Perl, '`%`' introduces a hash variable, but inserting a hash into a string doesn't make much sense, so interpolation via `%` is not supported.

- **Named Unicode Characters** If you have "**`use charnames ':full';`**" in the program, you can refer to Unicode characters by name using the `\N{`*name*`}` sequence. For instance, `\N{LATIN SMALL LETTER SHARP S}` matches "ß". The list of Unicode characters that Perl understands can be found in Perl's *unicore* directory, in the file *UnicodeData.txt*. This snippet shows the file's location:

  ```
  use Config;
  print "$Config{privlib}/unicore/UnicodeData.txt\n";
  ```

  It's easy to forget "**`use charnames ':full';`**", or the colon before '`full`', but if you do, `\N{…}` won't work. Also, `\N{…}` doesn't work if you use regex overloading, described later in this list.

- **Case-Folding Prefix** The special sequences **`\l`** and **`\u`** cause the character that follows to be made lowercase and uppercase, respectively. This is usually used just before variable interpolation to force the case on the first character brought in from the variable. For example, if the variable `$title` contains "`mr.`", the code `m/…\u$title…/` creates the regex `「…Mr.…」`. The same functionality is provided by the Perl functions `lcfirst()` and `ucfirst()`.

- **Case-Folding Span** The special sequences **`\L`** and **`\U`** cause characters that follow to be made lowercase and uppercase, respectively, until the end of the regex literal, or until the special sequence **`\E`**. For example, with the same `$title` as before, the code `m/…\U$title\E…/` creates the regex `「…MR.…」`. The same functionality is provided by the Perl functions `lc()` and `uc()`.

  You can combine a case-folding prefix with a case-folding span: the code `m/…\L\u$title\E…/` ensures `「…Mr.…」` regardless of the original capitalization.

- **Literal-Text Span** The sequence **`\Q`** "quotes" *regex* metacharacters (i.e., puts a backslash in front of them) until the end of the string, or until a **`\E`** sequence. It quotes *regex* metacharacters, but not quote *regex-literal* items like variable interpolation, **`\U`**, and, of course, the **`\E`** itself. Oddly, it also does not quote backslashes that are part of an unknown sequence, such as in `\F` or `\H`. Even with `\Q…\E`, such sequences still produce "unrecognized escape" warnings.

  In practice, these restrictions are not that big a drawback, as `\Q…\E` is normally used to quote interpolated text, where it properly quotes *all* metacharacters. For example, if `$title` contains "`Mr.`", the code `m/…\Q$title\E…/` creates the regex `「…Mr\.…」`, which is what you'd want if you wanted to match the *text* in `$title`, rather than the *regex* in `$title`.

  This is particularly useful if you want to incorporate user input into a regex. For example, `m/\Q$UserInput\E/i` does a case-insensitive search for the characters (as a string, not a regex) in `$UserInput`.

  The `\Q…\E` functionality is also provided by the Perl function `quotemeta()`.

- **Overloading** You can pre-process the literal parts of a regex literal in any way you like with *overloading*. It's an interesting concept, but one with severe limitations as currently implemented. Overloading is covered in detail, starting on page 341.

Picking your own regex delimiters

One of the most bizarre (yet, most useful) aspects of Perl's syntax is that you can pick your own delimiters for regex literals. The traditional delimiter is a forward slash, as with `m/…/`, `s/…/…/`, and `qr/…/`, but you can actually pick any non-alphanumeric, non-whitespace character. Some commonly used examples include:

```
m!…!              m{…}
m,…,              m<…>
s|…|…|            m[…]
qr#…#             m(…)
```

The four on the right are among the special-case delimiters:

- The four examples on the right side of the list above have different opening and closing delimiters, and may be nested (that is, may contain copies of the delimiters so long as the opens and closes pair up properly). Because parentheses and square brackets are so prevalent in regular expressions, `m(…)` and `m[…]` are probably not as appealing as the others. In particular, with the `/x` modifier, something such as the following becomes possible:

  ```
  m{
      regex  # comments
      here   # here
  }x;
  ```

  If one of these pairs is used for the regex part of a substitute, another pair (the same as the first, or, if you like, different) is used for the replacement string. Examples include:

  ```
  s{…}{…}
  s{…}!…!
  s<…>(…)
  s[…]/…/
  ```

  If this is done, you can put whitespace and comments between the two pairs of delimiters. More on the substitution operator's replacement string operand can be found on page 319.

- For the match operator only, a question mark as a delimiter has a little-used special meaning (suppress additional matches) discussed in the section on the match operator (☞ 308).

- As mentioned on page 288, a regex literal is parsed like a "regex-aware double-quoted string." If a single quote is used as the delimiter, however, those features are inhibited. With `m'…'`, variables are *not* interpolated, and the constructs that modify text on the fly (e.g., `\Q…\E`) do not work, nor does the `\N{…}` construct. `m'…'` might be convenient for a regex that has many `@`, to save having to escape them.

For the match operator only, the `m` may be omitted if the delimiter is a slash or a question mark. That is,

```
$text =~ m/…/;
$text =~  /…/;
```

are the same. My preference is to always explicitly use the `m`.

How Regex Literals Are Parsed

For the most part, one "just uses" the regex-literal features just discussed, without the need to understand the exact details of how Perl converts them to a raw regular expression. Perl is very good at being intuitive in this respect, but there are times when a more detailed understanding can help. The following lists the order in which processing appears to happen:

1. The closing delimiter is found, and the modifiers (such as `/i`, etc.) are read. The rest of the processing then knows if it's in `/x` mode.
2. Variables are interpolated.
3. If regex overloading is in effect, each part of the literal is given to the overload routine for processing. Parts are separated by interpolated variables; the values interpolated are not made available to overloading.

   If regex overloading is not in effect, `\N{…}` sequences are processed.
4. Case-folding constructs (e.g., `\Q…\E`) are applied.
5. The result is presented to the regex engine.

This describes how the processing appears to the programmer, but in reality, the internal processing done by Perl is quite complicated. Even step #2 must understand the regular-expression metacharacters, so as not to, for example, treat the underlined portion of `this$|that$` as a variable reference.

Regex Modifiers

Perl's regex operators allow *regex modifiers*, placed after the closing delimiter of the regex literal (like the **`i`** in `m/…/`**`i`**, `s/…/…/`**`i`**, or `qr/…/`**`i`**). There are five core modifiers that all regex operators support, shown in Table 7-3.

The first four, described in Chapter 3, can also be used within a regex itself as a mode-modifier (☞ 133) or mode-modified span (☞ 134). When used both within

Table 7-3: The Core Modifiers Available to All Regex Operators

`/i`	☞109	Ignore letter case during match
`/x`	☞110	Free-spacing and comments regex mode
`/s`	☞110	Dot-matches-all match mode
`/m`	☞111	Enhanced line anchor match mode
`/o`	☞348	Compile only once

the regex, and as part of one of the match operators, the in-regex versions take precedence for the part of the regex they control. (Another way to look at it is that once a modifier has been applied to some part of a regex, nothing can "unmodify" that part of a regex.)

The fifth core modifier, `/o`, has mostly to do with efficiency. It is discussed later in this chapter, starting on page 348.

If you need more than one modifier, group the letters together and place them in any order after the closing delimiter, whatever it might be.† Keep in mind that the slash is not part of the modifier—you can write **`m/<title>/i`** as **`m|<title>|i`**, or perhaps **`m{<title>}i`**, or even **`m<<title>>i`**. Nevertheless, when discussing modifiers, it's common to always write them with a slash, e.g., "the `/i` modifier."

Regex-Related Perlisms

A variety of general Perl concepts pertain to our study of regular expressions. The next few sections discuss:

- **Context** An important concept in Perl is that many functions and operators respond to the *context* they're used in. For example, Perl expects a scalar value as the conditional of a `while` loop, but a list of values as the arguments to a `print` statement. Since Perl allows expressions to "respond" to the context in which they're in, identical expressions in each case might produce wildly different results.
- **Dynamic Scope** Most programming languages support the concept of local and global variables, but Perl provides an additional twist with something known as *dynamic scoping*. Dynamic scoping temporarily "protects" a global variable by saving a copy of its value and automatically restoring it later. It's an intriguing concept that's important for us because it affects `$1` and other match-related variables.

† Because modifiers can appear in any order, a large portion of a programmer's time is spent adjusting the order to achieve maximum cuteness. For example, `learn/by/osmosis` is valid code (assuming you have a function called `learn`). The `osmosis` are the modifiers. Repeating modifiers is allowed, but meaningless (except for the substitution-operator's `/e` modifier, discussed later).

Expression Context

The notion of *context* is important throughout Perl, and in particular, to the match operator. An expression might find itself in one of three contexts, *list*, *scalar*, or *void*, indicating the type of value expected from the expression. Not surprisingly, a *list context* is one where a list of values is expected of an expression. A *scalar context* is one where a single value is expected. These two are very common and of great interest to our use of regular expressions. *Void context* is one in which no value is expected.

Consider the two assignments:

```
$s = expression one;
@a = expression two;
```

Because `$s` is a simple scalar variable (it holds a single value, not a list), it expects a simple scalar value, so the first expression, whatever it may be, finds itself in a scalar context. Similarly, because `@a` is an array variable and expects a list of values, the second expression finds itself in a list context. Even though the two expressions might be exactly the same, they might return completely different values, and cause completely different side effects while they're at it. Exactly what happens depends on each expression.

For example, the `localtime` function, if used in a list context, returns a list of values representing the current year, month, date, hour, etc. But if used in a scalar context, it returns a textual version of the current time along the lines of '`Mon Jan 20 22:05:15 2003`'.

As another example, an I/O operator such as `<MYDATA>` returns the next line of the file in a scalar context, but returns a list of all (remaining) lines in a list context.

Like `localtime` and the I/O operator, many Perl constructs respond to their context. The regex operators do as well — the match operator `m/···/`, for example, sometimes returns a simple true/false value, and sometimes a list of certain match results. All the details are found later in this chapter.

Contorting an expression

Not all expressions are natively context-sensitive, so Perl has rules about what happens when a general expression is used in a context that doesn't exactly match the type of value the expression normally returns. To make the square peg fit into a round hole, Perl "contorts" the value to make it fit. If a scalar value is returned in a list context, Perl makes a list containing the single value on the fly. Thus, **`@a = 42`** is the same as **`@a = (42)`**.

On the other hand, there's no general rule for converting a list to a scalar. If a literal list is given, such as with

```
$var = ($this, &is, 0xA, 'list');
```

the comma-operator returns the last element, `'list'`, for `$var`. If an array is given, as with **`$var = @array`**, the length of the array is returned.

Some words used to describe how other languages deal with this issue are *cast*, *promote*, *coerce*, and *convert*, but I feel they are a bit too consistent (boring?) to describe Perl's attitude in this respect, so I use "contort."

Dynamic Scope and Regex Match Effects

Perl's two types of storage (global and private variables) and its concept of *dynamic scoping* are important to understand in their own right, but are of particular interest to our study of regular expressions because of how after-match information is made available to the rest of the program. The next sections describe these concepts, and their relation to regular expressions.

Global and private variables

On a broad scale, Perl offers two types of variables: global and private. Private variables are declared using `my(…)`. Global variables are not declared, but just pop into existence when you use them. Global variables are always visible from anywhere and everywhere within the program, while private variables are visible, lexically, only to the end of their enclosing block. That is, the only Perl code that can directly access the private variable is the code that falls between the `my` declaration and the end of the block of code that encloses the `my`.

The use of global variables is normally discouraged, except for special cases, such as the myriad of special variables like `$1`, `$_`, and `@ARGV`. Regular user variables are global unless declared with `my`, even if they might "look" private. Perl allows the names of global variables to be partitioned into groups called *packages*, but the variables are still global. A global variable `$Debug` within the package `Acme::Widget` has a *fully qualified name* of `$Acme::Widget::Debug`, but no matter how it's referenced, it's still the same global variable. If you **`use strict;`**, all (non-special) globals must either be referenced via fully-qualified names, or via a name declared with `our` (`our` declares a *name*, not a new variable—see the Perl documentation for details).

Dynamically scoped values

Dynamic scoping is an interesting concept that few programming languages provide. We'll see the relevance to regular expressions soon, but in a nutshell, you can have Perl save a copy of the value of a global variable that you intend to

modify within a block, and restore the original copy automatically at the time when the block ends. Saving a copy is called *creating a new dynamic scope*, or *localizing*.

One reason that you might want to do this is to temporarily update some kind of global state that's maintained in a global variable. Let's say that you're using a package, `Acme::Widget`, and it provides a debugging flag via the global variable `$Acme::Widget::Debug`. You can temporarily ensure that debugging is turned on with code like:

```
    ⋮
{
    local($Acme::Widget::Debug) = 1; # Ensure it's turned on
    # work with Acme::Widget while debugging is on
      ⋮
}
# $Acme::Widget::Debug is now back to whatever it had been before
  ⋮
```

It's that extremely ill-named function `local` that creates a new dynamic scope. Let me say up front that *the call to `local` does not create a new variable*. `local` is an action, not a declaration. Given a global variable, `local` does three things:

1. Saves an internal copy of the variable's value
2. Copies a new value into the variable (either `undef`, or a value assigned to the `local`)
3. Slates the variable to have its original value restored when execution runs off the end of the block enclosing the `local`

This means that "local" refers only to how long any changes to the variable will last. The localized value lasts as long as the enclosing block is executing. Even if a subroutine is called from within that block, the localized value is seen. (After all, the variable is still a global variable.) The only difference from a non-localized global variable is that when execution of the enclosing block finally ends, the previous value is automatically restored.

An automatic save and restore of a global variable's value is pretty much all there is to `local`. For all the misunderstanding that has accompanied `local`, it's no more complex than the snippet on the right of Table 7-4 illustrates.

As a matter of convenience, you can assign a value to `local($SomeVar)`, which is exactly the same as assigning to `$SomeVar` in place of the `undef` assignment. Also, the parentheses can be omitted to force a scalar context.

As a practical example, consider having to call a function in a poorly written library that generates a lot of "Use of uninitialized value" warnings. You use Perl's `-w` option, as all good Perl programmers should, but the library author apparently didn't. You are exceedingly annoyed by the warnings, but if you can't change the

Table 7-4: The Meaning of local

Normal Perl	Equivalent Meaning

`{`
`  local($SomeVar); #` *save copy*
 	
`  $SomeVar = 'My Value';`
`  :`
`  :`
`  :`
`} #`*Value automatically restored*
	`{`
	`  my $TempCopy = $SomeVar;`
	`  $SomeVar = undef;`
	
	`  $SomeVar = 'My Value';`
	`  :`
	`  :`
	`  $SomeVar = $TempCopy;`
	`}`

library, what can you do short of stop using `-w` altogether? Well, you could set a `local` value of `$^W`, the in-code debugging flag (the variable name `^W` can be either the two characters, caret and 'W', or an actual control-W character):

```
{
    local $^W = 0; # Ensure warnings are off.
    UnrulyFunction(…);
}
# Exiting the block restores the original value of $^W.
```

The call to `local` saves an internal copy of the value of the global variable `$^W`, whatever it might be. Then that same `$^W` receives the new value of zero that we immediately scribble in. When `UnrulyFunction` is executing, Perl checks `$^W` and sees the zero we wrote, so doesn't issue warnings. When the function returns, our value of zero is still in effect.

So far, everything appears to work just as if `local` isn't used. However, when the block is exited right after the subroutine returns, the original value of `$^W` is restored. Your change of the value was local, in *time*, to the life of the block. You'd get the same effect by making and restoring a copy yourself, as in Table 7-4, but `local` conveniently takes care of it for you.

For completeness, let's consider what happens if I use `my` instead of `local`.† Using `my` creates a *new variable* with an initially undefined value. It is visible only within the lexical block it is declared in (that is, visible only by the code written between the `my` and the end of the enclosing block). It does not change, modify, or in any other way refer to or affect other variables, including any global variable of the same name that might exist. The newly created variable is not visible elsewhere in the program, including from within `UnrulyFunction`. In our example snippet, the new `$^W` is immediately set to zero but is never again used or referenced, so it's pretty much a waste of effort. (While executing `UnrulyFunction` and deciding whether to issue warnings, Perl checks the unrelated global variable `$^W`.)

† Perl doesn't allow the use of `my` with this special variable name, so the comparison is only academic.

A better analogy: clear transparencies

A useful analogy for `local` is that it provides a clear transparency (like used with an overhead projector) over a variable on which you scribble your own changes. You (and anyone else that happens to look, such as subroutines and signal handlers) will see the new values. They shadow the previous value until the point in time that the block is finally exited. At that point, the transparency is automatically removed, in effect, removing any changes that might have been made since the `local`.

This analogy is actually much closer to reality than saying "an internal copy is made." Using `local` doesn't actually make a copy, but instead puts your new value earlier in the list of those checked whenever a variable's value is accessed (that is, it shadows the original). Exiting a block removes any shadowing values added since the block started. Values are added manually, with `local`, but here's the whole reason we've been looking localization: **regex side-effect variables have their values dynamically scoped automatically**.

Regex side effects and dynamic scoping

What does dynamic scoping have to do with regular expressions? A lot. A number of variables like `$&` (refers to the text matched) and `$1` (refers to the text matched by the first parenthesized subexpression) are automatically set as a side effect of a successful match. They are discussed in detail in the next section. These variables have their value dynamically scoped *automatically* upon entry to every block.

To see the benefit of this design choice, realize that each call to a subroutine involves starting a new block, which means a new dynamic scope is created for these variables. Because the values before the block are restored when the block exits (that is, when the subroutine returns), the subroutine can't change the values that the caller sees.

As an example, consider:

```
if (m/(…)/)
{
    DoSomeOtherStuff();
    print "the matched text was $1.\n";
}
```

Because the value of `$1` is dynamically scoped automatically upon entering each block, this code snippet neither cares, nor needs to care, whether the function `DoSomeOtherStuff` changes the value of `$1` or not. Any changes to `$1` by the function are contained within the block that the function defines, or perhaps within a sub-block of the function. Therefore, they can't affect the value this snippet sees with the `print` after the function returns.

The automatic dynamic scoping is helpful even when not so apparent:

```
if ($result =~ m/ERROR=(.*)/) {
   warn "Hey, tell $Config{perladmin} about $1!\n";
}
```

The standard library module `Config` defines an associative array `%Config`, of which the member `$Config{perladmin}` holds the email address of the local Perlmaster. This code could be very surprising if `$1` were not automatically dynamically scoped, because `%Config` is actually a *tied* variable. That means any reference to it involves a behind-the-scenes subroutine call, and the subroutine within `Config` that fetches the appropriate value when `$Config{…}` is used invokes a regex match. That match lies between your match and your use of `$1`, so if `$1` were not dynamically scoped, it would be destroyed before you used it. As it is, any changes in `$1` during the `$Config{…}` processing are safely hidden by dynamic scoping.

Dynamic scoping versus lexical scoping

Dynamic scoping provides many rewards if used effectively, but haphazard dynamic scoping with `local` can create a maintenance nightmare, as readers of a program find it difficult to understand the increasingly complex interactions among the lexically disperse `local`, subroutine calls, and references to localized variables.

As I mentioned, the `my(…)` declaration creates a private variable with *lexical scope*. A private variable's lexical scope is the opposite of a global variable's global scope, but it has little to do with dynamic scoping (except that you can't `local` the value of a `my` variable). Remember, `local` is just an *action*, while `my` is both an action *and*, importantly, a declaration.

Special Variables Modified by a Match

A successful match or substitution sets a variety of global, read-only variables that are always automatically dynamically scoped. These values *never* change if a match attempt is unsuccessful, and are *always* set when a match is successful. When appropriate, they are set to the empty string (a string with no characters in it), or undefined (a "no value" value, similar to, yet testably distinct from, an empty string). Table 7-5 shows examples.

In more detail, here are the variables set after a match:

`$&` A copy of the text successfully matched by the regex. This variable (along with `$`` and `$'`, described next) is best avoided for performance reasons. (See the discussion on page 356.) `$&` is never undefined after a successful match, although it can be an empty string.

Table 7-5: Example Showing After-Match Special Variables

After the match of

```
                                  12                  2 3    4      4 31
  "Pi is 3.14159, roughly" =~ m/\b((tasty|fattening)|(\d+(\.\d*)?))\b/;
```

the following special variables are given the values shown.

Variable	Meaning	Value
`$``	Text before match	`Pi·is·`
`$&`	Text matched	`3.14159`
`$'`	Text after match	`,·roughly`
`$1`	Text matched within 1st set of parentheses	`3.14159`
`$2`	Text matched within 2nd set of parentheses	*undef*
`$3`	Text matched within 3rd set of parentheses	`3.14159`
`$4`	Text matched within 4th set of parentheses	`.14159`
`$+`	Text from highest-numbered `$1`, `$2`, etc.	`.14159`
`$^N`	Text from most recently closed `$1`, `$2`, etc.	`3.14159`
`@-`	Array of match-start indices into target text	`(6, 6, undef, 6, 7)`
`@+`	Array of match-end indices into target text	`(13, 13, undef, 13, 13)`

**`$``** A copy of the target text in front of (to the left of) the match's start. When used in conjunction with the `/g` modifier, you might wish `$`` to be the text from start of the *match attempt*, but it's the text from the start of the whole string, each time. `$`` is never undefined after a successful match.

`$'` A copy of the target text after (to the right of) the successfully matched text. `$'` is never undefined after a successful match. After a successful match, the string `"$`$&$'"` is always a copy of the original target text.†

`$1`, **`$2`**, **`$3`**, etc.

The text matched by the 1st, 2nd, 3rd, etc., set of capturing parentheses. (Note that `$0` is not included here—it is a copy of the script name and not related to regular expressions.) These are guaranteed to be undefined if they refer to a set of parentheses that doesn't exist in the regex, or to a set that wasn't actually involved in the match.

These variables are available after a match, including in the replacement operand of `s/…/…/`. They can also be used within the *code* parts of an embedded-code or dynamic-regex construct (☞327). Otherwise, it makes little sense to use them within the regex itself. (That's what ⌜`\1`⌟ and friends are for.) See "Using $1 Within a Regex?" on page 303.

The difference between ⌜`(\w+)`⌟ and ⌜`(\w)+`⌟ can be seen in how `$1` is set. Both regexes match exactly the same text, but they differ in what

† Actually, if the original target is undefined, but the match successful (unlikely, but possible), `"$`$&$'"` would be an empty string, not undefined. This is the only situation where the two differ.

subexpression falls within the parentheses. Matching against the string '`tubby`', the first one results in `$1` having the full '`tubby`', while the latter one results in it having only '`y`': with ⌜`(\w)+`⌟, the plus is outside the parentheses, so each iteration causes them to start capturing anew, leaving only the last character in `$1`.

Also, note the difference between ⌜`(x)?`⌟ and ⌜`(x?)`⌟. With the former, the parentheses and what they enclose are optional, so `$1` would be either '`x`' or undefined. But with ⌜`(x?)`⌟, the parentheses enclose a match — what is optional are the contents. If the overall regex matches, the contents matches something, although that something might be the nothingness ⌜`x?`⌟ allows. Thus, with ⌜`(x?)`⌟ the possible values of `$1` are '`x`' and an empty string. The following table shows some examples:

Sample Match	Resulting `$1`	Sample Match	Resulting `$1`
`"::" =~ m/:(A?):/`	empty string	`"::" =~ m/:(\w*):/`	empty string
`"::" =~ m/:(A)?:/`	undefined	`"::" =~ m/:(\w)*:/`	undefined
`":A:" =~ m/:(A?):/`	`A`	`":Word:" =~ m/:(\w*):/`	`Word`
`":A:" =~ m/:(A)?:/`	`A`	`":Word:" =~ m/:(\w)*:/`	`d`

When adding parentheses just for capturing, as was done here, the decision of which to use is dependent only upon the semantics you want. In these examples, since the added parentheses have no affect on the overall match (they all match the same text), the only differences among them is in the side effect of how `$1` is set.

`$+` This is a copy of the highest numbered `$1`, `$2`, etc. explicitly set during the match. This might be useful after something like

```
$url =~ m{
   href \s* = \s*    # Match the "href = " part, then the value . . .
   (?: "([^"]*)"     # a double-quoted value, or . . .
     | '([^']*)'     # a single-quoted value, or . . .
     | ([^'"<>]+) )  # an unquoted value.
}ix;
```

to access the value of the `href`. Without `$+`, you would have to check each of `$1`, `$2`, and `$3` and use the one that's not undefined.

If there are no capturing parentheses in the regex (or none are used during the match), it becomes undefined.

`$^N` A copy of the most-recently-closed `$1`, `$2`, etc. explicitly set during the match (i.e., the `$1`, `$2`, etc., associated with the final closing parenthesis). If there are no capturing parentheses in the regex (or none used during the match), it becomes undefined. A good example of its use is given starting on page 344.

@- and **@+**

These are arrays of starting and ending offsets (string indices) into the target text. They might be a bit confusing to work with, due to their odd names. The first element of each refers to the overall match. That is, the first element of `@-`, accessed with `$-[0]`, is the offset from the beginning of the target string to where the match started. Thus, after

```
$text = "Version 6 coming soon?";
   ⋮
$text =~ m/\d+/;
```

the value of `$-[0]` is `8`, indicating that the match started eight characters into the target string. (In Perl, indices are counted started at zero.)

The first element of `@+`, accessed with `$+[0]`, is the offset to the end of the match. With this example, it contains `9`, indicating that the overall match ended nine characters from the start of the string. So, using them together, **`substr($text, $-[0], $+[0] - $-[0])`** is the same as `$&` if `$text` has not been modified, but doesn't have the performance penalty that `$&` has (☞356). Here's an example showing a simple use of `@-`:

```
1 while $line =~ s/\t/' ' x (8 - $-[0] % 8)/e;
```

Given a line of text, it replaces tabs with the appropriate number of spaces.†

Subsequent elements of each array are the starting and ending offsets for captured groups. The pair `$-[1]` and `$+[1]` are the offsets into the target text where `$1` was taken, `$-[2]` and `$+[2]` for `$2`, and so on.

$^R This variable holds the resulting value of the most recently executed embedded-code construct, except that an embedded-code construct used as the *if* of a 「**(?** *if then* **|** *else***)**」 conditional (☞138) does not set `$^R`. When used within a regex (within the *code* parts of embedded-code and dynamic-regex constructs; ☞327), it is automatically localized to each part of the match, so values of `$^R` set by code that gets "unmatched" due to backtracking are properly forgotten. Put another way, it has the "most recent" value with respect to the match path that got the engine to the current location.

When a regex is applied repeatedly with the `/g` modifier, each iteration sets these variables afresh. That's why, for instance, you can use `$1` within the replacement operand of `s/…/…/g` and have it represent a new slice of text with each match.

† This tab-replacement snippet has the limitation that it works only with "traditional" western text. It doesn't produce correct results with wide characters like 枝, which is one character but takes up two spaces, nor some Unicode renditions of accented characters like à (☞107).

Using $1 within a regex?

The Perl man page makes a concerted effort to point out that ⌈\1⌋ is not available as a backreference outside of a regex. (Use the variable $1 instead.) The variable $1 refers to a string of static text matched during some previously completed successful match. On the other hand, ⌈\1⌋ is a true regex metacharacter that matches text similar to that matched within the first parenthesized subexpression *at the time that the regex-directed NFA reaches the* ⌈\1⌋. What it matches might change over the course of an attempt as the NFA tracks and backtracks in search of a match.

The opposite question is whether $1 and other after-match variables are available within a regex operand. They are commonly used within the *code* parts of embedded-code and dynamic-regex constructs (☞327), but otherwise make little sense within a regex. A $1 appearing in the "regex part" of a regex operand is treated exactly like any other variable: its value is interpolated before the match or substitution operation even begins. Thus, as far as the regex is concerned, the value of $1 has nothing to do with the current match, but rather is left over from some previous match.

The qr/···/ Operator and Regex Objects

Introduced briefly in Chapter 2 and Chapter 6 (☞76; 277), qr/···/ is a unary operator that takes a regex operand and returns a *regex object*. The returned object can then be used as a regex operand of a later match, substitution, or split, or can be used as a sub-part of a larger regex.

Regex objects are used primarily to encapsulate a regex into a unit that can be used to build larger expressions, and for efficiency (to gain control over exactly when a regex is compiled, discussed later).

As described on page 291, you can pick your own delimiters, such as qr{···} or qr!···!. It supports the core modifiers /i, /x, /s, /m, and /o.

Building and Using Regex Objects

Consider the following, with expressions adapted from Chapter 2 (☞76):

```
my $HostnameRegex = qr/[-a-z0-9]+(?:\.[-a-z0-9]+)*\.(?:com|edu|info)/i;

my $HttpUrl = qr{
   http:// $HostnameRegex \b   # Hostname
   (?:
        / [-a-z0-9_:\@&?=+,.!/~*'%\$]* # Optional path
            (?<![.,?!]) # Not allowed to end with [.,?!]
   )?
}ix;
```

The first line encapsulates our simplistic hostname-matching regex into a regular-expression object, and saves it to the variable `$HostnameRegex`. The next lines then use that in building a regex object to match an HTTP URL, saved to the variable `$HttpUrl`. Once constructed, they can be used in a variety of ways, such as

```
if ($text =~ $HttpUrl) {
   print "There is a URL\n";
}
```

to merely inspect, or perhaps

```
while ($text =~ m/($HttpUrl)/g) {
   print "Found URL: $1\n";
}
```

to find and display all HTTP URLs.

Now, consider changing the definition of `$HostnameRegex` to this, derived from Chapter 5 (☞ 205):

```
my $HostnameRegex = qr{
   # One or more dot-separated parts
   (?: [a-z0-9]\. | [a-z0-9][-a-z0-9]{0,61}[a-z0-9]\. )*
   # Followed by the final suffix part
   (?: com|edu|gov|int|mil|net|org|biz|info|...|aero|[a-z][a-z] )
}xi;
```

This is intended to be used in the same way as our previous version (for example, it doesn't have a leading `^` and trailing `$`, and has no capturing parentheses), so we're free to use it as a drop-in replacement. Doing so gives us a stronger `$HttpUrl`.

Match modes (or lack thereof) are very sticky

`qr/.../` supports the core modifiers described on page 292. Once a regex object is built, the match modes of the regex it represents can't be changed, even if that regex object is used inside a subsequent `m/.../` that has its own modifiers. For example, the following **does not** work:

```
my $WordRegex = qr/\b \w+ \b/; # Oops, missing the /x modifier!
   :
if ($text =~ m/^($WordRegex)/x) {
    print "found word at start of text: $1\n";
}
```

The `/x` modifiers are used here ostensibly to modify how `$WordRegex` is *applied*, but this does not work because the modifiers (or lack thereof) are locked in by the `qr/.../` when `$WordRegex` is *created*. So, the appropriate modifiers must be used at that time.

Here's a working version of the previous example:

```
my $WordRegex = qr/\b \w+ \b/x;   # This works!
    ⋮
if ($text =~ m/^($WordRegex)/) {
    print "found word at start of text: $1\n";
}
```

Now, contrast the original snippet with the following:

```
my $WordRegex = '\b \w+ \b';   # Normal string assignment
    ⋮
if ($text =~ m/^($WordRegex)/x) {
    print "found word at start of text: $1\n";
}
```

Unlike the original, this one works even though no modifiers are associated with `$WordRegex` when it is created. That's because in this case, `$WordRegex` is a normal variable holding a simple string that is interpolated into the `m/…/` regex literal. Building up a regex in a string is much less convenient than using regex objects, for a variety of reasons, including the problem in this case of having to remember that this `$WordRegex` must be applied with `/x` to be useful.

Actually, you can solve that problem even when using strings by putting the regex into a *mode-modified span* (☞ 134) when creating the string:

```
my $WordRegex = '(?x:\b \w+ \b)'; # Normal string assignment
    ⋮
if ($text =~ m/^($WordRegex)/) {
    print "found word at start of text: $1\n";
}
```

In this case, after the `m/…/` regex literal interpolates the string, the regex engine is presented with ⌜`^(?x:\b·\w+·\b)`⌟, which works the way we want.

In fact, this is what logically happens when a regex object is created, except that a regex object always explicitly defines the "on" or "off" for each of the `/i`, `/x`, `/m`, and `/s` modes. Using **`qr/\b·\w+·\b/x`** creates ⌜`(?x-ism:\b·\w+·\b)`⌟. Notice how the mode-modified span, ⌜`(?x-ism:…)`⌟, has `/x` turned on, while `/i`, `/s`, and `/m` are turned off. Thus, `qr/…/` always "locks in" each mode, whether given a modifier or not.

Viewing Regex Objects

The previous paragraph talks about how regex objects logically wrap their regular expression with mode-modified spans like ⌜`(?x-ism:…)`⌟. You can actually see this for yourself, because if you use a regex object where Perl expects a string, Perl kindly gives a textual representation of the regex it represents. For example:

```
% perl -e 'print qr/\b \w+ \b/x, "\n"'
(?x-ism:\b \w+ \b)
```

Here's what we get when we `print` the `$HttpUrl` from page 304:

```
(?ix-sm:
   http:// (?ix-sm:
   # One or more dot-separated parts⋯
   (?: [a-z0-9]\. | [a-z0-9][-a-z0-9]{0,61}[a-z0-9]\. )*
   # Followed by the final suffix part⋯
   (?: com|edu|gov|int|mil|net|org|biz|info|⋯|aero|[a-z][a-z] )
) \b          # hostname
   (?:
        / [-a-z0-9_:\@&?=+,.!/~*'%\$]* # Optional path
           (?<![.,?!]) # Not allowed to end with [.,?!]
   )?
)
```

The ability to turn a regex object into a string is very useful for debugging.

Using Regex Objects for Efficiency

One of the main reasons to use regex objects is to gain control, for efficiency reasons, of exactly when Perl compiles a regex to an internal form. The general issue of regex compilation was discussed briefly in Chapter 6, but the more complex Perl-related issues, including regex objects, are discussed in "Regex Compilation, the `/o` Modifier, `qr/⋯/`, and Efficiency" (☞348).

The Match Operator

The basic match

```
$text =~ m/regex/
```

is the core of Perl regular-expression use. In Perl, a regular-expression match is an *operator* that takes two *operands*, a target string operand and a regex operand, and returns a value.

How the match is carried out, and what kind of value is returned, depend on the context the match is used in (☞294), and other factors. The match operator is quite flexible—it can be used to test a regular expression against a string, to pluck data from a string, and even to parse a string part by part in conjunction with other match operators. While powerful, this flexibility can make mastering it more complex. Some areas of concern include:

- How to specify the regex operand
- How to specify match modifiers, and what they mean
- How to specify the target string to match against
- A match's side effects
- The value returned by a match
- Outside influences that affect the match

The general form of a match is:

```
StringOperand =~ RegexOperand
```

There are various shorthand forms, and it's interesting to note that each part is optional in one shorthand form or another. We'll see examples of all forms throughout this section.

Match's Regex Operand

The regex operand can be a regex literal or a regex object. (Actually, it can be a string or any arbitrary expression, but there is little benefit to that.) If a regex literal is used, match modifiers may also be specified.

Using a regex literal

The regex operand is most often a regex literal within `m/…/` or just `/…/`. The leading `m` is optional if the delimiters for the regex literal are forward slashes or question marks (delimiters of question marks are special, discussed in a bit). For consistency, I prefer to always use the `m`, even when it's not required. As described earlier, you can choose your own delimiters if the `m` is present (☞ 291).

When using a regex literal, you can use any of the core modifiers described on page 292. The match operator also supports two additional modifiers, `/g` and `/c`, discussed in a bit.

Using a regex object

The regex operand can also be a regex object, created with `qr/…/`. For example:

```
my $regex = qr/regex/;
  ⋮
if ($text =~ $regex) {
     ⋮
```

You can use `m/…/` with a regex object. As a special case, if the *only* thing within the "regex literal" is the interpolation of a regex object, it's exactly the same as using the regex object alone. This example's `if` can be written as:

```
if ($text =~ m/$regex/) {
     ⋮
```

This is convenient because it perhaps looks more familiar, and also allows you to use the `/g` modifier with a regex object. (You can use the other modifiers that `m/…/` supports as well, but they're meaningless in this case because they can never override the modes locked in a regex object ☞ 304.)

The default regex

If no regex is given, such as with `m//` (or with `m/$SomeVar/` where the variable `$SomeVar` is empty or undefined), Perl reuses the regular expression *most recently used successfully within the enclosing dynamic scope*. This used to be useful for efficiency reasons, but is now obsolete with the advent of regex objects (☞ 303).

Special match-once `?···?`

In addition to the special cases for the regex-literal delimiters described earlier, the match operator treats the question mark as a special delimiter. The use of a question mark as the delimiter (as with `m?···?`) enables a rather esoteric feature such that after the successfully `m?···?` matches once, it cannot match again until the function `reset` is called in the same package. Quoting from the Perl Version 1 manual page, this features was "a useful optimization when you only want to see the first occurrence of something in each of a set of files," but for whatever reason, I have never seen it used in modern Perl.

The question mark delimiters are a special case like the forward slash delimiters, in that the `m` is optional: `?···?` by itself is treated as `m?···?`.

Specifying the Match Target Operand

The normal way to indicate "this is the string to search" is using **`=~`**, as with **`$text =~ m/···/`**. Remember that **`=~`** is *not* an assignment operator, nor is it a comparison operator. It is merely a funny-looking way of linking the match operator with one of its operands. (The notation was adapted from awk.)

Since the whole "*expr* **`=~ m/···/`**" is an expression itself, you can use it wherever an expression is allowed. Some examples (each separated by a wavy line):

```
$text =~ m/···/;     # Just do it, presumably, for the side effects.
```

```
if ($text =~ m/···/) {
  # Do code if match is successful
   ⋮
```

```
$result = ( $text =~ m/···/ ); # Set $result to result of match against $text
$result =   $text =~ m/···/  ; # Same thing; =~ has higher precedence than =
```

```
   $copy = $text;               # Copy $text to $copy ...
   $copy           =~ m/···/;   # ... and perform match on $copy
( $copy = $text ) =~ m/···/;    # Same thing in one expression
```

The default target

If the target string is the variable `$_`, you can omit the "**`$_ =~`**" parts altogether. In other words, the default target operand is `$_`.

Something like

```
$text =~ m/regex/;
```

means "Apply *regex* to the text in `$text`, ignoring the return value but doing the side effects." If you forget the '~', the resulting

```
$text = m/regex/;
```

becomes "Apply *regex* to the text in `$_`, do the side effects, and return a true or false value that is then assigned to `$text`." In other words, the following are the same:

```
$text =         m/regex/;
$text = ($_ =~ m/regex/);
```

Using the default target string can be convenient when combined with other constructs that have the same default (as many do). For example, this is a common idiom:

```
while (<>)
{
   if (m/···/) {
     ⋮
   } elsif (m/···/) {
     ⋮
```

In general, though, relying on default operands can make your code less approachable by less experienced programmers.

Negating the sense of the match

You can also use `!~` instead of `=~` to logically negate the sense of the return value. (Return values and side effects are discussed soon, but with `!~`, the return value is always a simple true or false value.) The following are identical:

```
if ($text !~ m/···/)

if (not $text =~ m/···/)

unless ($text =~ m/···/)
```

Personally, I prefer the middle form. With any of them, the normal side effects, such as the setting of `$1` and the like, still happen. `!~` is merely a convenience in an "if this doesn't match" situation.

Different Uses of the Match Operator

You can always use the match operator as if it returns a simple true/false indicating the success of the match, but there are ways you can get additional information about a successful match, and to work in conjunction with other match operators. How the match operator works depends primarily on the *context* in which it's used (☞ 294), and whether the `/g` modifier has been applied.

Normal "does this match?"—scalar context without /g

In a scalar context, such as the test of an `if`, the match operator returns a simple true or false:

```
if ($target =~ m/…/) {
    # . . . processing after successful match . . .
     ⋮
} else {
    # . . . processing after unsuccessful match . . .
     ⋮
}
```

You can also assign the result to a scalar for inspection later:

```
my $success  =  $target =~ m/…/;
  ⋮
if ($success) {
  ⋮
}
```

Normal "pluck data from a string"—list context, without /g

A list context without `/g` is the normal way to pluck information from a string. The return value is a list with an element for each set of capturing parentheses in the regex. A simple example is processing a date of the form `69/8/31`, using:

```
my ($year, $month, $day)  =  $date =~ m{^ (\d+) / (\d+) / (\d+) $}x;
```

The three matched numbers are then available in the three variables (and `$1`, `$2`, and `$3` as well). There is one element in the return-value list for each set of capturing parentheses, or an empty list upon failure.

It is possible for a set of capturing parentheses to not participate in the final success of a match. For example, one of the sets in `m/(this)|(that)/` is guaranteed not to be part of the match. Such sets return the undefined value `undef`. If there are no sets of capturing parentheses to begin with, a successful list-context match without `/g` returns the list `(1)`.

A list context can be provided in a number of ways, including assigning the results to an array, as with:

```
my @parts  =  $text =~ m/^(\d+)-(\d+)-(\d+)$/;
```

If you're assigning to just one scalar variable, take care to provide a list context to the match if you want the captured parts instead of just a Boolean indicating the success. Compare the following tests:

```
my ($word)   =  $text =~ m/(\w+)/;
my $success  =  $text =~ m/(\w+)/;
```

The parentheses around the variable in the first example cause its `my` to provide a list context to the assignment (in this case, to the match). The lack of parentheses

in the second example provides a scalar context to the match, so `$success` merely gets a true/false result.

This example shows a convenient idiom:

```
if ( my ($year, $month, $day) = $date =~ m{^ (\d+) / (\d+) / (\d+) $}x ) {
    # Process for when we have a match: $year and such are available
} else {
    # here if no match . . .
}
```

The match is in a list context (provided by the "`my (…) =`"), so the list of variables is assigned their respective `$1`, `$2`, etc., if the match is successful. However, once that's done, since the whole combination is in the scalar context provided by the `if` conditional, Perl must contort the list to a scalar. To do that, it takes the number of items in the list, which is conveniently zero if the match wasn't successful, and non-zero (i.e., true) if it was.

"Pluck all matches"—list context, with the /g modifier

This useful construct returns a list of all text matched within capturing parentheses (or if there are no capturing parentheses, the text matched by the whole expression), not only for one match, as in the previous section, but for all matches in the string.

A simple example is the following, to fetch all integers in a string:

```
my @nums  =  $text =~ m/\d+/g;
```

If `$text` contains an IP address like '`64.156.215.240`', `@nums` then receives four elements, '`64`', '`156`', '`215`', and '`240`'. Combined with other constructs, here's an easy way to turn an IP address into an eight-digit hexadecimal number such as '`409cd7f0`', which might be convenient for creating compact log files:

```
my $hex_ip = join '', map { sprintf("%02x", $_) } $ip =~ m/\d+/g;
```

You can convert it back with a similar technique:

```
my $ip = join '.', map { hex($_) } $hex_ip =~ m/../g
```

As another example, to match all floating-point numbers on a line, you might use:

```
my @nums  =  $text =~ m/\d+(?:\.\d+)?|\.\d+/g;
```

The use of non-capturing parentheses here is very important, since adding capturing ones changes what is returned. Here's an example showing how one set of capturing parentheses can be useful:

```
my @Tags  =  $Html =~ m/<(\w+)/g;
```

This sets `@Tags` to the list of HTML tags, in order, found in `$Html`, assuming it contains no stray '<' characters.

Here's an example with multiple sets of capturing parentheses: consider having the entire text of a Unix mailbox alias file in a single string, where logical lines look like:

```
alias  Jeff      jfriedl@regex.info
alias  Perlbug   perl5-porters@perl.org
alias  Prez      president@whitehouse.gov
```

To pluck an alias and full address from one of the logical lines, you can use **`m/^alias\s+(\S+)\s+(.+)/m`** (without `/g`). In a list context, this returns a list of two elements, such as **`('Jeff', 'jfriedl@regex.info')`**. Now, to match all such sets, add `/g`. This returns a list like:

```
( 'Jeff', 'jfriedl@regex.info', 'Perlbug',
  'perl5-porters@perl.org', 'Prez', 'president@whitehouse.gov' )
```

If the list happens to fit a key/value pair pattern as in this example, you can actually assign it directly to an associative array. After running

```
my %alias  =  $text =~ m/^alias\s+(\S+)\s+(.+)/mg;
```

you can access the full address of '`Jeff`' with `$alias{Jeff}`.

Iterative Matching: Scalar Context, with /g

A scalar-context `m/⋯/g` is a special construct quite different from the others. Like a normal `m/⋯/`, it does just one match, but like a list-context `m/⋯/g`, it pays attention to where previous matches occurred. Each time a scalar-context `m/⋯/g` is reached, such as in a loop, it finds the "next" match. If it fails, it resets the "current position," causing the next application to start again at the beginning of the string.

Here's a simple example:

```
$text = "WOW! This is a SILLY test.";

$text =~ m/\b([a-z]+\b)/g;
print "The first all-lowercase word: $1\n";

$text =~ m/\b([A-Z]+\b)/g;
print "The subsequent all-uppercase word: $1\n";
```

With both scalar matches using the `/g` modifier, it results in:

```
The first all-lowercase word: is
The subsequent all-uppercase word: SILLY
```

The two scalar-`/g` matches work together: the first sets the "current position" to just after the matched lowercase word, and the second picks up from there to find the first uppercase word *that follows*. The `/g` is required for either match to pay attention to the "current position," so if *either* didn't have `/g`, the second line would refer to '`WOW`'.

A scalar context `/g` match is quite convenient as the conditional of a `while` loop. Consider:

```
while ($ConfigData =~ m/^(\w+)=(.*)/mg) {
    my($key, $value) = ($1, $2);
      ⋮
}
```

All matches are eventually found, but the body of the `while` loop is executed between the matches (well, *after* each match). Once an attempt fails, the result is false and the `while` loop finishes. Also, upon failure, the `/g` state is reset, which means that the next `/g` match starts over at the start of the string.

Compare

```
while ($text =~ m/(\d+)/) { # dangerous!
    print "found: $1\n";
}
```

and:

```
while ($text =~ m/(\d+)/g) {
    print "found: $1\n";
}
```

The only difference is `/g`, but it's a huge difference. If `$text` contained, say, our earlier IP example, the second prints what we want:

```
found: 64
found: 156
found: 215
found: 240
```

The first, however, prints "`found: 64`" over and over, forever. Without the `/g`, the match is simply "find the first `(\d+)` in `$text`," which is '`64`' no matter how many times it's checked. Adding the `/g` to the scalar-context match turns it into "find the *next* `(\d+)` in `$text`," which finds each number in turn.

The "current match location" and the `pos()` function

Every string in Perl has associated with it a "current match location" at which the transmission first attempts the match. It's a property of the string, and not associated with any particular regular expression. When a string is created or modified, the "current match location" starts out at the beginning of the string, but when a `/g` match is successful, it's left at the location where the match ended. The next time a `/g` match is applied to the string, the match begins inspecting the string at that same "current match location."

You have access to the target string's "current match location" via the `pos(···)` function. For example:

```
my $ip = "64.156.215.240";
while ($ip =~ m/(\d+)/g) {
   printf "found '$1' ending at location %d\n", pos($ip);
}
```

This produces:

```
found '64' ending at location 2
found '156' ending at location 6
found '215' ending at location 10
found '240' ending at location 14
```

(Remember, string indices are zero-based, so "location 2" is just before the 3rd character into the string.) After a successful `/g` match, `$+[0]` (the first element of `@+` ☞302) is the same as the `pos` of the target string.

The default argument to the `pos()` function is the same default argument for the match operator: the `$_` variable.

Pre-setting a string's `pos`

The real power of `pos()` is that you can write to it, to tell the regex engine where to start the next match (if that next match uses `/g`, of course). For example, the web server logs I work with at Yahoo! are in a custom format that contains 32 bytes of fixed-width data, followed by the page being requested, followed by other information. One way to pick out the page is to use `⌈^.{32}⌋` to skip over the fixed-width data:

```
if ($logline =~ m/^.{32}(\S+)/) {
    $RequestedPage = $1;
}
```

This brute-force method isn't elegant, and forces the regex engine to work to skip the first 32 bytes. That's less efficient and less clear than doing it explicitly ourself:

```
pos($logline) = 32; # The page starts at the 32nd character, so start the next match there . . .
if ($logline =~ m/(\S+)/g) {
    $RequestedPage = $1;
}
```

This is better, but isn't quite the same. It has the regex *start* where we want it to start, but doesn't require a match *at that position* the way the original does. If for some reason the 32nd character can't be matched by `⌈\S⌋`, the original version correctly fails, but the new version, without anything to anchor it to a particular position in the string, is subject to the transmission's bump-along. Thus, it could return, in error, a match of `⌈\S+⌋` from later in the string. Luckily, the next section shows that this is an easy problem to fix.

Using ⌈\G⌋

⌈\G⌋ is the "anchor to where the previous match ended" metacharacter. It's exactly what we need to solve the problem in the previous section:

```
pos($logline) = 32; # The page starts at the 32nd character, so start the next match there . . .
if ($logline =~ m/\G(\S+)/g) {
    $RequestedPage = $1;
}
```

⌈\G⌋ tells the transmission "don't bump-along with this regex — if you can't match successfully right away, fail."

There are discussions of ⌈\G⌋ in previous chapters: see the general discussion in Chapter 3 (☞ 128), and the extended example in Chapter 5 (☞ 212).

Note that Perl's ⌈\G⌋ is restricted in that it works predictably only when it is the first thing in the regex, and there is no top-level alternation. For example, in Chapter 6 when the CSV example is being optimized (☞ 271), the regex begins with ⌈\G(?:^|,)⋯⌋. Because there's no need to check for ⌈\G⌋ if the more restrictive ⌈^⌋ matches, you might be tempted to change this to ⌈(?:^|\G,)⋯⌋. Unfortunately, this doesn't work in Perl; the results are unpredictable.†

"Tag-team" matching with /gc

Normally, a failing m/⋯/g match attempt resets the target string's pos to the start of the string, but adding the /c modifier to /g introduces a special twist, causing a failing match to *not* reset the target's pos. (/c is never used without /g, so I tend to refer to it as /gc.)

m/⋯/gc is most commonly used in conjunction with ⌈\G⌋ to create a "lexer" that tokenizes a string into its component parts. Here's a simple example to tokenize the HTML in variable $html:

```
while (not $html =~ m/\G\z/gc) # While we haven't worked to the end . . .
{
  if    ($html =~ m/\G( <[^>]+>   )/xgc) { print "TAG: $1\n"            }
  elsif ($html =~ m/\G( &\w+;     )/xgc) { print "NAMED ENTITY: $1\n"   }
  elsif ($html =~ m/\G( &\#\d+;   )/xgc) { print "NUMERIC ENTITY: $1\n" }
  elsif ($html =~ m/\G( [^<>&\n]+ )/xgc) { print "TEXT: $1\n"           }
  elsif ($html =~ m/\G  \n         /xgc) { print "NEWLINE\n"            }
  elsif ($html =~ m/\G( .         )/xgc) { print "ILLEGAL CHAR: $1\n"   }
  else {
      die "$0: oops, this shouldn't happen!";
  }
}
```

† This *would* work with most other flavors that support ⌈\G⌋, but even so, I would generally not recommend using it, as the optimization gains by having ⌈\G⌋ at the start of the regex usually outweigh the small gain by not testing ⌈\G⌋ an extra time (☞ 245).

The bold part of each regex matches one type of HTML construct. Each is checked in turn starting from the current position (due to `/gc`), but can match *only* at the current position (due to ⌈`\G`⌋). The regexes are checked in order until the construct at that current position has been found and reported. This leaves `$html`'s `pos` at the start of the next token, which is found during the next iteration of the loop.

The loop ends when `m/\G\z/gc` is able to match, which is when the current position (⌈`\G`⌋) has worked its way to the very end of the string (⌈`\z`⌋).

An important aspect of this approach is that one of the tests *must* match each time through the loop. If one doesn't (and if we don't abort), there would be an infinite loop, since nothing would be advancing or resetting `$html`'s `pos`. This example has a final *else* clause that will never be invoked as the program stands now, but if we were to edit the program (as we will soon), we could perhaps introduce a mistake, so keeping the *else* clause is prudent. As it is now, if the data contains a sequence we haven't planned for (such as '<>'), it generates one warning message per unexpected character.

Another important aspect of this approach is the ordering of the checks, such as the placement of ⌈`\G(.)`⌋ as the last check. Or, consider extending this application to recognize `<script>` blocks with:

```
$html =~ m/\G ( <script[^>]*>.*?</script> )/xgcsi
```

(Wow, we've used five modifiers!) To work properly, this must be inserted into the program *before* the currently-first ⌈`<[^>]+>`⌋. Otherwise, ⌈`<[^>]+>`⌋ would match the opening `<script>` tag "out from under" us.

There's a somewhat more advanced example of `/gc` in Chapter 3 (☞ 130).

Pos-related summary

Here's a summary of how the match operator interacts with the target string's `pos`:

Type of match	Where match starts	`pos` upon success	`pos` upon failure
`m/…/`	start of string (`pos` ignored)	reset to `undef`	reset to `undef`
`m/…/g`	starts at target's `pos`	set to end of match	reset to `undef`
`m/…/gc`	starts at target's `pos`	set to end of match	left unchanged

Also, modifying a string in any way causes its `pos` to be reset to `undef` (which is the initial value, meaning the start of the string).

The Match Operator's Environmental Relations

The following sections summarize what we've seen about how the match operator influences the Perl environment, and vice versa.

The match operator's side effects

Often, the side effects of a successful match are more important than the actual return value. In fact, it is quite common to use the match operator in a void context (i.e., in such a way that the return value isn't even inspected), just to obtain the side effects. (In such a case, it acts as if given a scalar context.) The following summarizes the side effects of a *successful* match attempt:

- After-match variables like `$1` and `@+` are set for the remainder of the current scope (☞ 299).
- The *default regex* is set for the remainder of the current scope (☞ 308).
- If `m?…?` matches, it (the specific `m?…?` operator) is marked as unmatchable, at least until the next call of `reset` in the same package (☞ 308).

Again, these side effects occur only with a match that is successful—an unsuccessful match attempt has no influence on them. However, the following side effects happen with *any* match attempt:

- `pos` is set or reset for the target string (☞ 313).
- If `/o` is used, the regex is "fused" to the operator so that re-evaluation does not occur (☞ 352).

Outside influences on the match operator

What a match operator does is influenced by more than just its operands and modifiers. This list summarizes the outside influences on the match operator:

context
: The context that a match operator is applied in (scalar, array, or void) has a large influence on how the match is performed, as well as on its return value and side effects.

`pos(…)`
: The `pos` of the target string (set explicitly or implicitly by a previous match) indicates where in the string the next **`/g`**-governed match should begin. It is also where `\G` matches.

default regex
: The default regex is used if the provided regex is empty (☞ 308).

`study`
: It has no effect on what is matched or returned, but if the target string has been studied, the match might be faster (or slower). See "The Study Function" (☞ 359).

`m?…?` and `reset`
: The invisible "has/hasn't matched" status of `m?…?` operators is set when `m?…?` matches or `reset` is called (☞ 308).

Keeping your mind in context (and context in mind)

Before leaving the match operator, I'll put a question to you. Particularly when changing among the `while`, `if`, and `foreach` control constructs, you really need to keep your wits about you. What do you expect the following to print?

```
while ("Larry Curly Moe" =~ m/\w+/g) {
   print "WHILE stooge is $&.\n";
}
print "\n";

if ("Larry Curly Moe" =~ m/\w+/g) {
   print "IF stooge is $&.\n";
}
print "\n";

foreach ("Larry Curly Moe" =~ m/\w+/g) {
   print "FOREACH stooge is $&.\n";
}
```

It's a bit tricky. ❖ Turn the page to check your answer.

The Substitution Operator

Perl's substitution operator **`s/···/···/`** extends a match to a full match-and-replace. The general form is:

```
$text =~ s/regex/replacement/modifiers
```

In short, the text first matched by the regex operand is replaced by the value of the replacement operand. If the `/g` modifier is used, the regex is repeatedly applied to the text following the match, with additional matched text replaced as well.

As with the match operator, the target text operand and the connecting **`=~`** are optional if the target is the variable `$_`. But unlike the match operator's `m`, the substitution's `s` is never optional.

We've seen that the match operator is fairly complex—how it works, and what it returns, is dependent upon the context it's called in, the target string's `pos`, and the modifiers used. In contrast, the substitution operator is simple: it always returns the same information (an indication of the number of substitutions done), and the modifiers that influence how it works are easy to understand.

You can use any of the core modifiers described on page 292, but the substitution operator also supports two additional modifiers: `/g` and, described in a bit, `/e`.

The Replacement Operand

With the normal `s/…/…/`, the replacement operand immediately follows the regex operand, using a total of three instances of the delimiter rather than the two of `m/…/`. If the regex uses balanced delimiters (such as `<…>`), the replacement operand then has its own independent pair of delimiters (yielding a total of four). For example, `s{…}{…}` and `s[…]/…/` and `s<…>'…'` are all valid. In such cases, the two sets may be separated by whitespace, and if so, by comments as well. Balanced delimiters are commonly used with `/x` or `/e`:

```
$text =~ s{
    ...some big regex here, with lots of comments and such...
} {
    ...a Perl code snippet to be evaluated to produce the replacement text...
}ex;
```

Take care to separate in your mind the regex and replacement operands. The regex operand is parsed in a special regex-specific way, with its own set of special delimiters (☞ 291). The replacement operand is parsed and evaluated as a normal double-quoted string. The evaluation happens after the match (and with `/g`, after each match), so `$1` and the like are available to refer to the proper match slice.

There are two situations where the replacement operand is not parsed as a double-quoted string:

- When the replacement operand's delimiters are single quotes, it is parsed as a single-quoted string, which means that no variable interpolation is done.
- If the `/e` modifier (discussed in the next section) is used, the replacement operand is parsed like a little Perl script instead of like a double-quoted string. The little Perl script is executed after each match, with its result being used as the replacement.

The /e Modifier

The `/e` modifier causes the replacement operand to be evaluated as a Perl code snippet, as if with **`eval {…}`**. The code snippet's syntax is checked to ensure it's valid Perl when the script is loaded, but the code is evaluated afresh after each match. After each match, the replacement operand is evaluated in a scalar context, and the result of the code is used as the replacement. Here's a simple example:

```
$text =~ s/-time-/localtime/ge;
```

This replaces occurrences of `-time-` with the results of calling Perl's `localtime` function in a scalar context (which returns a textual representation of the current time, such as "`Wed Sep 25 18:36:51 2002`").

Since the evaluation is done after each match, you can refer to the text just matched with the after-match variables like `$1`. For example, special characters

Quiz Answer

❖ *Answer to the question on page 318.*

The question snippets on page 318 produce:

```
WHILE stooge is Larry.
WHILE stooge is Curly.
WHILE stooge is Moe.

IF stooge is Larry.

FOREACH stooge is Moe.
FOREACH stooge is Moe.
FOREACH stooge is Moe.
```

Note that if the `print` within the `foreach` loop had referred to `$_` rather than `$&`, its results would have been the same as the `while`'s. In this `foreach` case, however, the result returned by the `m/…/g`, **`('Larry', 'Curly', 'Moe')`**, goes unused. Rather, the side effect `$&` is used, which almost certainly indicates a programming mistake, as the side effects of a list-context `m/…/g` are not often useful.

that might not otherwise be allowed in a URL can be encoded using `%` followed by their two-digit hexadecimal representation. To encode all non-alphanumerics this way, you can use

```
$url =~ s/([^a-zA-Z0-9])/sprintf('%%%02x', ord($1))/ge;
```

and to decode back to the original, you can use:

```
$url =~ s/%([0-9a-f][0-9a-f])/pack("C", hex($1))/ige;
```

In short, **`sprintf('%%%02x', ord(`***character***`))`** converts characters to their numeric URL representation, while **`pack("C",`** *value***`)`** does the opposite; consult your favorite Perl documentation for more information.

Multiple uses of /e

Normally, repeating a modifier with an operator doesn't hurt (except perhaps to confuse the reader), but repeating the `/e` modifier actually changes how the replacement is done. Normally, the replacement operand is evaluated once, but if more than one '`e`' is given, the results of the evaluation are themselves evaluated as Perl, over and over, for as many extra '`e`' as are provided. This is perhaps useful mostly for an Obfuscated Perl Contest.

Still, it can be useful. Consider interpolating variables into a string manually (such as if the string is read from a configuration file). That is, you have a string that looks like '`… $var …`' and you want to replace the substring '`$var`' with the value of the variable `$var`.

A simple approach uses:

```
$data =~ s/(\$[a-zA-Z_]\w*)/$1/eeg;
```

Without any `/e`, this would simply replace the matched '`$var`' with itself, which is not very useful. With one `/e`, it evaluates the code `$1`, yielding '`$var`', which again, effectively replaces the matched text with itself (which is again, not very useful). But with two `/e`, that '`$var`' is itself evaluated, yielding its contents. Thus, this mimics the interpolation of variables.

Context and Return Value

Recall that the match operator returns different values based upon the particular combination of context and `/g`. The substitution operator, however, has none of these complexities—it always returns either the number of substitutions performed or, if none were done, an empty string.

Conveniently, when interpreted as a Boolean (such as for the conditional of an `if`), the return value is taken as true if any substitutions are done, false if not.

The Split Operator

The multifaceted `split` operator (often called a *function* in casual conversation) is commonly used as the converse of a list-context `m/…/g` (☞311). The latter returns text matched by the regex, while a `split` with the same regex returns text *separated* by matches. The normal match **`$text =~ m/:/g`** applied against a `$text` of '**`IO.SYS:225558:95-10-03:-a-sh:optional`**', returns the four-element list

```
(':', ':', ':', ':')
```

which doesn't seem useful. On the other hand, **`split(/:/, $text)`** returns the five-element list:

```
('IO.SYS', '225558', '95-10-03', '-a-sh', 'optional')
```

Both examples reflect that `:` matches four times. With `split`, those four matches partition a copy of the target into five chunks, which are returned as a list of five strings.

That example splits the target string on a single character, but you can split on any arbitrary regular expression. For example,

```
@Paragraphs = split(m/\s*<p>\s*/i, $html);
```

splits the HTML in `$html` into chunks, at `<p>` or `<P>`, surrounded by optional whitespace. You can even split on locations, as with

```
@Lines = split(m/^/m, $lines);
```

to break a string into its logical lines.

In its most simple form with simple data like this, `split` is as easy to understand as it is useful. However, there are many options, special cases, and special

situations that complicate things. Before getting into the details, let me show two particularly useful special cases:

- The special match operand `//` causes the target string to be split into its component characters. Thus, `split(//, "short test")` returns a list of ten elements: `("s", "h", "o", ···, "s", "t")`.
- The special match operand `" "` (a normal string with a single space) causes the target string to be split on whitespace, similar to using `m/\s+/` as the operand, except that any leading and trailing whitespace are ignored. Thus, `split(" ", "  a short  test  ")` returns the strings 'a', 'short', and 'test'.

These and other special cases are discussed a bit later, but first, the next sections go over the basics.

Basic Split

`split` is an operator that looks like a function, and takes up to three operands:

```
split(match operand, target string, chunk-limit operand)
```

The parentheses are optional. Default values (discussed later in this section) are provided for operands left off the end.

`split` is always used in a list context. Common usage patterns include:

```
($var1, $var2, $var3, ···) = split(···);
@array = split(···);
for my $item (split(···)) {
    ⋮
}
```

Basic match operand

The match operand has several special-case situations, but it is normally the same as the regex operand of the match operator. That means that you can use `/···/` and `m{···}` and the like, a regex object, or any expression that can evaluate to a string. Only the core modifiers described on page 292 are supported.

If you need parentheses for grouping, be sure to use the `(?:···)` non-capturing kind. As we'll see in a few pages, the use of capturing parentheses with `split` turns on a very special feature.

Target string operand

The target string is inspected, but is never modified by `split`. The content of `$_` is the default if no target string is provided.

Basic chunk-limit operand

In its primary role, the chunk-limit operand specifies a limit to the number of chunks that `split` partitions the string into. With the sample data from the first example, **`split(/:/, $text, 3)`** returns:

```
( 'IO.SYS', '225558', '95-10-03:-a-sh:optional' )
```

This shows that `split` stopped after `/:/` matched twice, resulting in the requested three-chunk partition. It could have matched additional times, but that's irrelevant because of this example's chunk limit. The limit is an upper bound, so no more than that many elements will ever be returned (unless the regex has capturing parentheses, which is covered in a later section). You may still get fewer elements than the chunk limit; if the data can't be partitioned enough to begin with, nothing extra is produced to "fill the count." With our example data, **`split(/:/, $text, 99)`** still returns only a five-element list. However, there is an important difference between **`split(/:/, $text)`** and **`split(/:/, $text, 99)`** which does not manifest itself with this example — keep this in mind when the details are discussed later.

Remember that the *chunk*-limit operand refers to the *chunks* between the matches, not to the number of matches themselves. If the limit were to refer to the matches themselves, the previous example with a limit of three would produce

```
('IO.SYS', '225558', '95-10-03', '-a-sh:optional')
```

which is not what actually happens.

One comment on efficiency: let's say you intended to fetch only the first few fields, such as with:

```
($filename, $size, $date) = split(/:/, $text);
```

As a performance enhancement, Perl stops splitting after the fields you've requested have been filled. It does this by automatically providing a chunk limit of one more than the number of items in the list.

Advanced split

`split` can be simple to use, as with the examples we've seen so far, but it has three special issues that can make it somewhat complex in practice:

- Returning empty elements
- Special regex operands
- A regex with capturing parentheses

The next sections cover these in detail.

Returning Empty Elements

The basic premise of `split` is that it returns the text separated by matches, but there are times when that returned text is an empty string (a string of length zero, e.g., `""`). For example, consider

```
@nums = split(m/:/, "12:34::78");
```

This returns

```
("12", "34", "", "78")
```

The regex `/:/` matches three times, so four elements are returned. The empty third element reflects that the regex matched twice in a row, with no text in between.

Trailing empty elements

Normally, trailing empty elements are *not* returned. For example,

```
@nums = split(m/:/, "12:34::78:::");
```

sets `@nums` to the same four elements

```
("12", "34", "", "78")
```

as the previous example, even though the regex was able to match a few extra times at the end of the string. By default, `split` does not return empty elements at the end of the list. However, you can have `split` return all trailing elements by using an appropriate chunk-limit operand . . .

The chunk-limit operand's second job

In addition to possibly limiting the number of chunks, any non-zero chunk-limit operand also preserves trailing empty items. (A chunk limit given as zero is exactly the same as if no chunk limit is given at all.) If you don't want to limit the number of chunks returned, but do want to leave trailing empty elements intact, simply choose a very large limit. Or, better yet, use `-1`, because a negative chunk limit is taken as an arbitrarily large limit: **`split(/:/, $text, -1)`** returns all elements, including any trailing empty ones.

At the other extreme, if you want to remove *all* empty items, you could put **`grep {length}`** before the `split`. This use of `grep` lets pass only list elements with non-zero lengths (in other words, elements that aren't empty):

```
my @NonEmpty = grep { length } split(/:/, $text);
```

Special matches at the ends of the string

A match at the very beginning normally produces an empty element:

```
@nums = split(m/:/, ":12:34::78");
```

That sets `@nums` to:

```
("", "12", "34", "", "78")
```

The initial empty element reflects the fact that the regex matched at the beginning of the string. However, as a special case, if the regex doesn't actually match any text when it matches at the start or end of the string, leading and/or trailing empty elements are *not* produced. A simple example is **`split(/\b/, "a simple test")`**, which can match at the six marked locations in '`a simple test`'. Even though it matches six times, it doesn't return seven elements, but rather only the five elements: **`("a", " ", "simple", " ", "test")`**. Actually, we've already seen this special case, with the **`@Lines = split(m/^/m, $lines)`** example on page 321.

Split's Special Regex Operands

`split`'s match operand is normally a regex literal or a regex object, as with the match operator, but there are some special cases:

- An empty regex for `split` does not mean "Use the current default regex," but to split the target string into a list of characters. We saw this before at the start of the `split` discussion, noting that **`split(//, "short test")`** returns a list of ten elements: **`("s", "h", "o", ···, "s", "t")`**.
- A match operand that is a *string* (**not** a regex) consisting of exactly one space is a special case. It's almost the same as `/\s+/`, except that leading whitespace is skipped. This is all meant to simulate the default input-record-separator splitting that awk does with its input, although it can certainly be quite useful for general use.

  If you'd like to keep leading whitespace, just use `m/\s+/` directly. If you'd like to keep trailing whitespace, use `-1` as the chunk-limit operand.
- If no regex operand is given, a string consisting of one space (the special case in the previous point) is used as the default. Thus, a raw `split` without any operands is the same as **`split(' ', $_, 0)`**.
- If the regex ⌈`^`⌋ is used, the `/m` modifier (for the enhanced line-anchor match mode) is automatically supplied for you. (For some reason, this does not happen for ⌈`$`⌋.) Since it's so easy to just use `m/^/`**`m`** explicitly, I would recommend doing so, for clarity. Splitting on `m/^/m` is an easy way to break a multiline string into individual lines.

Split has no side effects

Note that a `split` match operand often *looks* like a match operator, but it has none of the side effects of one. The use of a regex with `split` doesn't affect the default regex for later match or substitution operators. The variables `$&`, `$'`, `$1`,

and so on are not set or otherwise affected by a `split`. A `split` is completely isolated from the rest of the program with respect to side effects.†

Split's Match Operand with Capturing Parentheses

Capturing parentheses change the whole face of `split`. When they are used, the returned list has additional, independent elements interjected for the item(s) captured by the parentheses. This means that some or all text normally *not* returned by `split` is now included in the returned list.

For example, as part of HTML processing, `split(/(<[^>]*>)/)` turns

```
...and <B>very <FONT color=red>very</FONT> much</B> effort...
```

into:

```
( '...and ', '<B>', 'very ', '<FONT color=red>',
  'very', '</FONT>', ' much', '</B>', ' effort...' )
```

With the capturing parentheses removed, `split(/<[^>]*>/)` returns:

```
( '...and ', 'very ', 'very', ' much', ' effort...' )
```

The added elements do not count against a chunk limit. (The chunk limit limits the chunks that the original string is partitioned into, not the number of elements returned.)

If there are multiple sets of capturing parentheses, multiple items are added to the list with each match. If there are sets of capturing parentheses that don't contribute to a match, `undef` elements are inserted for them.

Fun with Perl Enhancements

Many regular-expression concepts that are now available in other languages were first made available only in Perl. Examples include non-capturing parentheses, lookahead, (and later, lookbehind), free-spacing mode, (most modes, actually — and with them comes ⌈`\A`⌋, ⌈`\z`⌋, and ⌈`\Z`⌋), atomic grouping, ⌈`\G`⌋, and the conditional construct. However, these are no longer Perl specific, so they are all covered in the main chapters of this book.

Still, Perl developers remain innovative, so there are some major concepts available at this time only in Perl. One of the most interesting is the ability to execute arbitrary code *during the match attempt*. Perl has long featured strong integration of regular expressions into code, but this brings integration to a whole new level.

† Actually, there is one side effect remaining from a feature that has been deprecated for many years, but has not actually been removed from the language yet. If `split` is used in a scalar or void context, it writes its results to the `@_` variable (which is also the variable used to pass function arguments, so be careful not to use `split` in these contexts by accident). **`use warnings`** or the `-w` command-line argument warns you if `split` is used in either context.

We'll continue with a short overview about this and other innovations available currently only in Perl, followed by the details.

The *dynamic regex* construct ⌈**(??{** *perl code* **})**⌋

Each time this construct is reached during the application of the regex in which it's found, the *perl code* is executed. The result of that execution (either a regex object or a string that's then interpreted as a regex) is applied right then, as part of the current match.

This simple example ⌈`^(\d+)(??{ "X{$1}" })$`⌋ is shown with the dynamic regex construct underlined. Overall, this regex matches a number at the beginning of the line, followed by exactly that many 'X' until the end of the line. It matches '3XXX' and '12XXXXXXXXXXXX', for example, but not '3X' or '7XXXX'. If we trace though the '3XXX' example, the leading ⌈`(\d+)`⌋ part matches '3XXX', setting `$1` to '3'. The regex engine then reaches the dynamic regex construct, which executes the code `"X{$1}"`, resulting in the value 'X{3}'. This is then interpreted as ⌈`X{3}`⌋, and applied as part of the current regex (matching the '3XXX'). Once that's done, the trailing ⌈`$`⌋ then matches at '3XXX', resulting in an overall match.

As we'll see in the examples that follow, a dynamic regex is particularly useful for matching arbitrarily nested constructs.

The *embedded-code* construct ⌈**(?{** *arbitrary perl code* **})**⌋

Like the dynamic regex construct, this construct also executes the Perl code each time it's reached during the application of a regex, but this construct is more general in that the code doesn't need to return any specific result. Usually, the return value is not even used. (But in case it is needed later in the same regex, the return value is available in the `$^R` variable ☞302).

There's one case where the value produced by the code is used: when an embedded-code construct is used as the *if* of an ⌈**(?** *if then* **|** *else***)**⌋ conditional (☞138). In this case, the result is interpreted as a Boolean value, upon which either the *then* or *else* part will be applied.

Embedded code can be used for many things, but it's particularly useful for debugging. Here's a simple example that displays a message every time the regex is actually applied, with the embedded-code construct underlined:

```
"have a nice day" =~ m{
   (?{ print "Starting match.\n" })
   \b(?: the | an | a )\b
}x;
```

The regex matches fully just once in this test, but the message is shown six times, reflecting that the regex was at least partially applied by the transmission at the five character positions prior to the sixth time, at which point it matches fully.

Regex-literal overloading

Regex-literal overloading lets you add your own custom pre-processing of regex literals, before they're given to the regex engine. You can use this to effectively add features to Perl's regex flavor. For example, Perl doesn't have separate start-of-word and end-of-word metacharacters (it has a catch-all `\b` word boundary), but you might want to have it recognize `\<` and `\>`, converting these constructs behind the scenes to ones Perl does know.

Regex overloading has some important limitations that severely restrict its usefulness. We'll look at this, as well as examples like the `\<` and `\>` idea, later in this section.

Force match of single byte

One other feature I should mention in this list is that the `\C` metacharacter matches one *byte*, even if that byte is just one of several that might encode a single *character*. This is dangerous—its misuse can cause internal errors, so it shouldn't be used unless you really know what you're doing. I can't think of a good use for it, so I won't mention it further.

When working with Perl code embedded within a regex (either in a dynamic regex construct or an embedded-code construct), it's best to use only global variables until you understand the important issue related to `my` variables discussed starting on page 338.

Using a Dynamic Regex to Match Nested Pairs

A dynamic regex's main use is to allow a regex to match arbitrarily nested constructs (something long thought to be impossible with regular expressions). Its quintessential example is to match content with arbitrarily nested parentheses. To see how a dynamic regex is useful for this, let's first look at why it's not possible with traditional constructs.

This simple regex matches a parenthesized run of text: `\( ( [^()] )* \)`. It doesn't allow parentheses within the outer parentheses, so it doesn't allow any nesting (that is, it supports zero levels of nesting). We can put it into a regex object and use it like this:

```
my $Level0 = qr/ \(  ( [^()] )*  \) /x; # Parenthesized text
  ⋮
if ($text =~ m/\b( \w+$Level0 )/x) {
   print "found function call: $1\n";
}
```

This would match "`substr($str, 0, 3)`", but not "`substr($str, 0, (3+2))`" because it has nested parentheses. Let's expand our regex to handle it. That means accommodating one level of nesting.

Allowing one level of nesting means allowing parenthesized text within the outer parentheses. So, we need to expand on the subexpression that matches between them, which is currently `[^()]`, by adding a subexpression that matches parenthesized text. Well, we just created that: `$Level0` holds such a regex. Using it, we can create the next level:

```
my $Level0 = qr/ \(  ( [^()]          )*  \) /x; # Parenthesized text
my $Level1 = qr/ \(  ( [^()]| $Level0 )*  \) /x; # One level of nesting
```

The `$Level0` here is the same as before; what's new is its use in building `$Level1`, which matches its own set of parentheses, *plus* those of `$Level0`. That's one level of nesting.

To match another level, we can use the same approach, creating a `$Level2` that uses `$Level1` (which still uses `$Level0`):

```
my $Level0 = qr/ \(  ( [^()]           )*  \) /x; # Parenthesized text
my $Level1 = qr/ \(  ( [^()] | $Level0 )*  \) /x; # One level of nesting
my $Level2 = qr/ \(  ( [^()] | $Level1 )*  \) /x; # Two levels of nesting
```

We can continue this indefinitely:

```
my $Level3 = qr/ \(  ( [^()] | $Level2 )*  \) /x; # Three levels of nesting
my $Level4 = qr/ \(  ( [^()] | $Level3 )*  \) /x; # Four levels of nesting
my $Level5 = qr/ \(  ( [^()] | $Level4 )*  \) /x; # Five levels of nesting
    ⋮
```

Figure 7-1 shows the first few levels graphically.

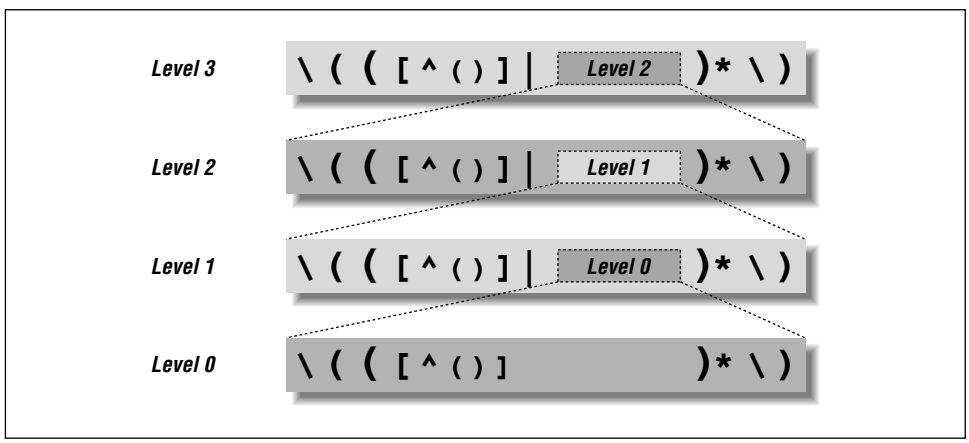

Figure 7-1: Matching a few levels of parentheses

It's interesting to see the result of all those levels. Here's what `$Level3` boils down to:

```
\(([^()]|\(([^()]|\(([^()]|\(([^()])*\))*\))*\))*\)
```

Wow, that's ugly.

Luckily, we don't have to interpret it directly (that's the regex engine's job). The approach with the `Level` variables is easy enough to work with, but its drawback is that nesting is limited to however many `$Level` variables we build. This approach doesn't allow us to match to an *arbitrary level*. (Murphy's Law being what it is, if we happen to pick *X* levels to support, we'll run into data with *X+1* levels of nesting.)

Luckily, we can use a dynamic regex to handle nesting to an arbitrary level. To get there, realize that each of the `$Level` variables beyond the first is constructed identically: when it needs to match an additional level of nesting, it includes the `$Level` variable below it. But if the `$Level` variables are all the same, it could just as well include the `$Level` above it. In fact, if they're all the same, it could just include *itself*. If it could somehow include itself when it wanted to match another level of nesting, it would recursively handle *any* level of nesting.

And that's just what we can do with a dynamic regex. If we create a regex object comparable to one of the `$Level` variables, we can refer to it from within a dynamic regex. (A dynamic-regex construct can contain arbitrary Perl code, so long as its results can be interpreted as a regular expression; Perl code that merely returns a pre-existing regex object certainly fits the bill.) If we put our `$Level`-like regex object into `$LevelN`, we can refer to it with ⌜`(??{ $LevelN })`⌟, like this:

```
my $LevelN; # This must be predeclared because it's used in its own definition.
$LevelN = qr/ \(( [^()] | (??{ $LevelN }) )* \) /x;
```

This matches arbitrarily nested parenthesized text, and can be used just like `$Level0` was used earlier:

```
if ($text =~ m/\b( \w+$LevelN )/x) {
   print "found function call: $1\n";
}
```

Phew! It's not necessarily easy to wrap one's brain around this, but once it "clicks," it's a valuable tool.

Now that we have the basic approach worked out, I'd like to make a few tweaks for efficiency's sake. I'll replace the capturing parentheses with atomic grouping (there's no need to capture, nor to backtrack), and once that's done, I can change ⌜`[^()]`⌟ to ⌜`[^()]+`⌟ for added efficiency. (Don't make this change without using atomic grouping, or you'll set yourself up for a neverending match ☞ 226.)

Finally, I'd like to move the ⌜`\(`⌟ and ⌜`\)`⌟ so that they directly surround the dynamic regex. This way, the dynamic regex construct isn't invoked by the engine until it's sure that there's something for it to match. Here's the revised version:

```
$LevelN = qr/ (?> [^()]+ | \( (??{ $LevelN }) \)  )* /x;
```

Since this no longer has outer ⌜`\(`⋯`\)`⌟, we need to include them ourselves when invoking `$LevelN`.

As a side effect of that, we have the flexibility to apply it where there *may* be sets of parentheses, not just where there *are* sets of parentheses:

```
if ($text =~ m/\b( \w+ \( $LevelN \) )/x) {
   print "found function call: $1\n";
}
```

```
if (not $text =~ m/^ $LevelN $/x) {
   print "mismatched parentheses!\n";
}
```

You can see another example of `$LevelN` in action on page 343.

Using the Embedded-Code Construct

The embedded-code construct is particularly useful for regex debugging, and for accumulating information about a match while it's happening. The next few pages walk through a series of examples that eventually lead to a method for mimicking POSIX match semantics. The journey there is perhaps more interesting than the actual destination (unless you need POSIX match semantics, of course) because of the useful techniques and insight we gain along the way.

We'll start with some simple regex debugging techniques.

Using embedded code to display match-time information

This code:

```
"abcdefgh" =~ m{
  (?{ print "starting match at [$`|$']\n" })
  (?:d|e|f)
}x;
```

produces:

```
starting match at [|abcdefgh]
starting match at [a|bcdefgh]
starting match at [ab|cdefgh]
starting match at [abc|defgh]
```

The embedded-code construct is the first thing in the regex, and so executes

```
print "starting match at [$`|$']\n"
```

whenever the regex starts a new match attempt. The displayed string uses the `$`` and `$'` variables (☞300)† to print the target text being matched, with '`|`' inserted to mark the current location in the match (which in this case is where the match attempt is starting). From the result, you can tell that the regex was applied four times by the transmission (☞148) before it was successful.

† Normally, I recommend against using the special match variables `$``, `$&`, and `$'`, as they can inflict a major efficiency penalty on the entire program (☞356), but they're fine for temporary debugging.

In fact, if we were to add

```
(?{ print "matched at [$`<$&>$']\n" })
```

just before the end of the regex, it would show the match:

```
matched at [abc<d>efgh]
```

Now, compare the first example with the following, which is identical except that the "main" regex is now `[def]` rather than `(?:d|e|f)`:

```
"abcdefgh" =~ m{
  (?{  print "starting match at [$`|$']\n" })
  [def]
}x;
```

In theory, the results should be identical, yet this produces only:

```
starting match at [abc|defgh]
```

Why the difference? Perl is smart enough to apply the *initial class discrimination* optimization (☞ 246) to the regex with `[def]`, thereby allowing the transmission to bypass attempts it felt were obviously destined to fail. As it turns out, it was able to bypass all attempts except the one that resulted in a match, and the embedded-code construct allows us to see that happen.

panic: top_env

If you're working with embedded code or a dynamic regex, and your program suddenly ends with an unceremonial

```
panic: top_env
```

it is likely due to a syntax error in the code part of the regex. Perl currently doesn't handle certain kinds of broken syntax well, and the panic is the result. The solution, of course, is to correct the syntax.

Using embedded code to see all matches

Perl has a Traditional NFA engine, so it stops the moment a match is found, even though there may be additional possible matches. With the clever use of embedded code, we can trick Perl into showing us *all* possible matches. To see how, let's revisit the silly '`oneself`' example from page 177:

```
"oneselfsufficient" =~ m{
    one(self)?(selfsufficient)?
   (?{ print "matched at [$`<$&>$']\n" })
}x;
```

As might be expected, this displays

```
matched at [<oneself>sufficient]
```

indicating that '`oneself`sufficient' had been matched at that point in the regex.

It's important to realize that despite the "`matched`" in the message, the `print` is not actually showing "the match," but rather the match *to that point*. The distinction is academic with this example because the embedded-code construct is the last thing in the regex. We know that the regex does indeed finish the moment the embedded-code construct has finished, reporting that same result as the actual match.

What if we added `(?!)` just after the embedded-code construct? `(?!)` is a negative lookahead that always fails. When it fails just after the embedded code is processed (just after a "matched" message is printed), it forces the engine to backtrack in search of a (new) match. The failure is forced after every "match" is printed, so we end up exploring every path to a match, and thus see all possible matches:

```
matched at [<oneself>sufficient]
matched at [<oneselfsufficient>]
matched at [<one>selfsufficient]
```

What we've done ensures that the overall match attempt actually fails, but in doing so we've got the regex engine to report all the possible matches. Without the `(?!)`, Perl returns the first match found, but with it, we can see the remaining permutations.

With that in mind, what do you think the following prints?

```
"123" =~ m{
   \d+
   (?{ print "matched at [$`<$&>$']\n" })
   (?!)
}x;
```

It displays:

```
matched at [<123>]
matched at [<12>3]
matched at [<1>23]
matched at [1<23>]
matched at [1<2>3]
matched at [12<3>]
```

Hopefully at least the first three were expected, but the rest might be unexpected if you're not on your toes. The `(?!)` forces backtracking and the eventual appearance of the 2nd and 3rd lines. When the attempt at the start of the line fails, the transmission reapplies the regex again starting just before the 2nd character. (Chapter 4 explains this in great detail.) The 4th and 5th lines shown are from that second attempt, and the last line shown is from the third attempt.

So, adding the `(?!)` really does cause it to show *all* possible matches, not just all of them from a particular starting point. It may be useful to see only the possible matches from a particular starting point; we'll look into that in a bit.

Finding the longest match

Now, instead of showing all the matches, let's find and save the longest match. We can do this by using a variable to keep track of the longest match seen so far and comparing each new "almost match" against it. Here is the solution with the '`oneself`' example:

```
$longest_match = undef; # We'll keep track of the longest match here

"oneselfsufficient" =~ m{
   one(self)?(selfsufficient)?
   (?{
      # Check to see if the current match ($&) is the longest so far
      if (not defined($longest_match)
          or
          length($&) > length($longest_match))
      {
          $longest_match = $&;
      }
   })
   (?!) # Force failure so we'll backtrack to find further "matches"
}x;

# Now report the accumulated result, if any
if (defined($longest_match)) {
   print "longest match=[$longest_match]\n";
} else {
   print "no match\n";
}
```

Not surprisingly, this shows '`longest match=[oneselfsufficient]`'. That bit of embedded code is pretty long, and something we'll likely use in the future, so let's encapsulate it and the `(?!)` into their own regex object:

```
my $RecordPossibleMatch = qr{
   (?{
      # Check to see if the current match ($&) is the longest so far
      if (not defined($longest_match)
          or
          length($&) > length($longest_match))
      {
          $longest_match = $&;
      }
   })
   (?!) # Force failure so we'll backtrack to find further "matches"
}x;
```

Here's a simple example that finds '9938', the longest match *overall*:

```
$longest_match = undef; # We'll keep track of the longest match here
"800-998-9938" =~ m{  \d+  $RecordPossibleMatch  }x;
# Now report the accumulated result, if any
if (defined($longest_match)) {
   print "longest match=[$longest_match]\n";
} else {
   print "no match\n";
}
```

Finding the longest-leftmost match

Now that we know how to find the longest match overall, let's restrict it to finding the longest-*leftmost* match. That just happens to be the match that a POSIX NFA would find (☞177). To accomplish this, we need to disable the transmission's bump-ahead *if* we've seen a match so far. That way, once we find the first match, normal backtracking still brings us to any other matches available from the same starting location (allowing us to keep track of the longest match), but the disabled bump-ahead inhibits the finding of matches that start later in the string.

Perl doesn't give us direct hooks into the transmission, so we can't disable the bump-ahead directly, but we can get the same effect by not allowing the regex to proceed past the start if `$longest_match` is already defined. The test for that is ⌈**(?{ defined $longest_match})**⌋, but that alone not enough, since it's just a test. The key to using the results of the test lies in a *conditional*.

Using embedded code in a conditional

To have the regex engine respond to the results of our test, we use the test as the *if* of an ⌈**(?** *if* *then* **|** *else***)**⌋ conditional (☞138). Since we want the regex to stop if the test is true, we use a fail-now `(?!)` as the *then* part. (We don't need an *else* part, so we just omit it.) Here's a regex object that encapsulates the conditional:

```
my $BailIfAnyMatch = qr/(?(?{ defined $longest_match})(?!))/;
```

The *if* part is underlined, and the *then* part is shown in bold. Here it is in action, combined with the `$RecordPossibleMatch` defined on the facing page:

```
"800-998-9938" =~ m{ $BailIfAnyMatch  \d+  $RecordPossibleMatch  }x;
```

This finds '800', the POSIX "longest of all leftmost matches" match.

Using `local` in an Embedded-Code Construct

The use of `local` within an embedded-code construct takes on special meaning. Understanding it requires a good understanding of *dynamic scoping* (☞295) and of the "crummy analogy" from the Chapter 4's discussion of how a regex-directed NFA engine goes about its work (☞158). The following contrived (and, as we'll see, flawed) example helps to illustrate why, without a lot of extraneous clutter. It

checks to see if a line is composed of only ⌈\w+⌋ and ⌈\s+⌋, but counts how many of the ⌈\w+⌋ are really ⌈\d+\b⌋:

```
my $Count = 0;

$text =~ m{
    ^ (?> \d+ (?{ $Count++ }) \b | \w+ | \s+ )* $
}x;
```

When this is matched against a string like '123 abc 73 9271 xyz', the $Count variable is left with a value of three. However, when applied to '123 abc 73xyz' it's left with a value of two, even though it should be left with a value of just one. The problem is that $Count is updated after matching '73', something that is matched by ⌈\d+⌋ but later "unmatched" via backtracking because the subsequent ⌈\b⌋ can't match. The problem arises because the code executed via the embedded-code construct is not somehow "unexecuted" when its part of the regex is "unmatched" via backtracking.

In case you have any confusion with the use of ⌈(?>···)⌋ atomic grouping (☞ 137) and the backtracking going on here, I'll mention that the atomic grouping is used to prevent a neverending-match (☞ 269), and does not affect backtracking *within* the construct, only backtracking *back into* the construct after it's been exited. So the ⌈\d+⌋ is free to be "unmatched" if the subsequent ⌈\b⌋ cannot match.

The easy solution for this contrived example is to put the ⌈\b⌋ before incrementing $Count, to ensure that it is incremented only when it won't be undone. However, I'd like to show a solution using local, to illustrate its effect within Perl executed during the application of a regex. With that in mind, consider this new version:

```
our $Count = 0;

$text =~ m{
 ^ (?> \d+ (?{ local($Count) = $Count + 1 }) \b | \w+ | \s+ )* $
}x;
```

The first change to notice is that $Count changed from a my variable to a global one (if you **use strict**, as I always recommend, you can't use an unqualified global variable unless you "declare" it with Perl's our declarator).

The other change is that the increment of $Count has been localized. Here's the key behavior: *when a variable is localized within a regex, the original value is replaced (the new value is lost) if the code with the* local *is "unmatched" because of backtracking*. So, even though local($Count) = $Count + 1 is executed after '73' is matched by ⌈\d+⌋, changing $Count from one to two, that change is "localized to the success of the path" that the regex is on when local is called. When the ⌈\b⌋ fails, the regex engine logically backtracks to before the local, and $Count reverts to its original value of one. And that's the value it ends up having when the end of the regex is eventually reached.

Interpolating Embedded Perl

As a security measure, Perl doesn't normally allow an embedded-code construct ⌜`(?{…})`⌟ or a dynamic-subexpression construct ⌜`(??{…})`⌟ to be interpolated into the regex from a string variable. (They are allowed, though, from a regex object, as with `$RecordPossibleMatch` on page 334.) That is,

```
m{   (?{ print "starting\n" })  some regex…  }x;
```

is allowed, but

```
my $ShowStart = '(?{ print "starting\n" })';
  ⋮
m{ $ShowStart some regex…  }x;
```

is not. This limitation is imposed because it has long been common to include user input as part of a regex, and the introduction of these constructs suddenly allowing such a regex to run arbitrary code creates a huge security hole. So, the default is that it's disallowed.

If you'd like to allow this kind of interpolation, the declaration:

```
use re 'eval';
```

lifts the restriction. (With different arguments, the `use re` pragma can also be used for debugging; ☞361.)

Sanitizing user input for interpolation

If you use this and do allow user input to be interpolated, be sure that it has no embedded-Perl or dynamic-regex constructs. You can do this by checking against ⌜`\(\s*\?+[p{]`⌟. If this matches the input, it's not safe to use in a regex. The ⌜`\s*`⌟ is needed because the `/x` modifier allows spaces after the opening parentheses. (I'd think that they shouldn't be allowed there, but they are.) The plus quantifies ⌜`\?`⌟ so that both constructs are recognized. Finally, the `p` is included to catch the now-deprecated ⌜`(?p{…})`⌟ construct, the forerunner of ⌜`(??{…})`⌟.

I think it would be useful if Perl supported a modifier of some sort that allowed or prohibited embedded code on a per-regex or subexpression basis, but until one is introduced, you'll have to check for it yourself, as described above.

So, `local` is required to keep `$Count` consistent until the end of the regex. If we were to put ⌜`(?{ print "Final count is $Count.\n" })`⌟ at the end of the regex, it would report the proper count. Since we want to use `$Count` after the match, we need to save it to a non-localized variable at some point before the match officially ends. This is because all values that had been localized during the match are lost when the match finishes.

Here's an example:

```
my $Count = undef;
our $TmpCount = 0;

$text =~ m{
 ^ (?> \d+ (?{ local($TmpCount) = $TmpCount + 1 }) \b | \w+ | \s+ )* $
 (?{ $Count = $TmpCount }) # Save the "ending" $Count to a non-localized variable
}x;
if (defined $Count) {
    print "Count is $Count.\n";
} else {
    print "no match\n";
}
```

This seems like a lot of work for something so simple, but again, this is a contrived example designed just to show the mechanics of localized variables within a regex. We'll see practical use in "Mimicking Named Capture" on page 344.

A Warning About Embedded Code and my Variables

If you have a my variable declared *outside* a regex, but refer to it from *inside* regex embedded code, you must be very careful about a subtle issue with Perl's variable binding that has a very unsubtle impact. Before describing the issue, I'll note up front that if you use only global variables within regex embedded code, you don't have to worry about this issue, and you can safely skip this section. Warning: this section is not light reading.

This contrived example illustrates the problem:

```
sub CheckOptimizer
{
    my $text  = shift; # The first argument is the text to check.
    my $start = undef; # We'll note here where the regex is first applied.

    my $match = $text =~ m{
      (?{ $start = $-[0] if not defined $start}) # Save the first starting position
      \d # This is the regex being tested
    }x;

    if (not defined $start) {
        print "The whole match was optimized away.\n";
        if ($match) {
            # This can't possibly happen!
            print "Whoa, but it matched! How can this happen!?\n";
        }
    } elsif ($start == 0) {
        print "The match start was not optimized.\n";
    } else {
        print "The optimizer started the match at character $start.\n"
    }
}
```

This code has three my variables, but only one, $start, is related to this issue (the others are not referenced from within embedded code, so are not at issue). It

works by first setting `$start` to the undefined value, then applying a match in which the leading component is an embedded-code construct that sets `$start` to the starting location of the attempt, but only if it hasn't already been set. The "starting location of the attempt" is derived from `$-[0]` (the first element of `@-` ☞302).

So, when this function is called with

```
CheckOptimizer("test 123");
```

the result is:

```
The optimizer started the match at character 5.
```

That's okay, but if we invoke the exact same call again, the second time shows:

```
The whole match was optimized away.
Whoa, but it matched! How can this happen!?
```

Even though the text checked by the regex is the same (as is the regex itself, for that matter), the result is different, and seems to be wrong. Why? The problem is that the second time through, the `$start` that the embedded code is updating is the one that existed the first time through, when the regex was compiled. The `$start` that the rest of the function uses is actually a new variable created afresh when the `my` is executed at the start of each function call.

The key to this issue is that `my` variables in embedded code are "locked in" (*bound*, in programming terminology) to *the specific instance* of the `my` variable that is active *at the time* the regex is compiled. (Regex compilation is discussed in detail starting on page 348.) Each time `CheckOptimizer` is called, a *new instance* of `$start` is created, but for esoteric reasons, the `$start` inside the embedded code still refers to the first instance that is now long gone. Thus, the instance of `$start` that the rest of the function uses doesn't receive the value ostensibly written to it within the regex.

This type of instance binding is called a *closure*, and books like *Programming Perl* and *Object Oriented Perl* discuss why it's a valuable feature of the language. There is debate in the Perl community, however, as to just how much of a "feature" it is in this case. To most people, it's very unintuitive.

The solution is to not refer to `my` variables from within a regex unless you know that the regex literal will be compiled at least as often as the `my` instances are refreshed. For example, the `my` variable `$NestedStuffRegex` is used within the `SimpleConvert` subroutine in the listing on page 346, but we know this is not a problem because there's only ever one instance of `$NestedStuffRegex`. Its `my` is not in a function or a loop, so it's created just once when the script is loaded, with that same instance existing until the program ends.

Matching Nested Constructs with Embedded Code

The example on page 328 shows how to match arbitrarily nested pairs using a dynamic regex. That's generally the easiest way to do it, but it's instructive to see a method using only embedded-code constructs, so I'd like to show it to you here.

The approach is simply this: keep a count of how many open parentheses we've seen that have not yet been closed, and allow a closing parenthesis only if there are outstanding opens. We'll use embedded code to keep track of the count as we match through the text, but before looking at that, let's look at a (not yet working) skeleton the expression:

```
my $NestedGuts = qr{
  (?>
    (?:
       # Stuff not parenthesis
         [^()]+
       # An opening parenthesis
       |  \(
       # A closing parenthesis
       |  \)
    )*
  )
}x;
```

The atomic grouping is required for efficiency, to keep the 「`( [⋯]+ | ⋯ )*`」 from becoming a neverending match (☞226) if `$NestedGuts` is used as part of some larger expression that could cause backtracking. For example, if we used it as part of `m/^\( $NestedGuts \)$/x` and applied it to '`(this is missing the close`', it would track and backtrack for a long time if atomic grouping didn't prune the redundant states.

To incorporate the counting, we need these four steps:

❶ Before beginning, the count must be initialized to zero:

```
(?{ local $OpenParens = 0 })
```

❷ When an open parenthesis is seen, we increment the count to indicate that one more set of parentheses has yet to balance.

```
(?{ $OpenParens++ })
```

❸ When a close parenthesis is seen, we check the count, and if it's currently positive, we decrement the count to recognize that one less set remains unbalanced. On the other hand, if the count is zero, we can't allow the match to continue (because the close parenthesis does not balance with an open), so we apply 「`(?!)`」 to force failure:

```
(?(?{ $OpenParens }) (?{ $OpenParens-- }) | (?!) )
```

This uses an 「**(?** *if* *then* **|** *else***)**」 conditional (☞138), with an embedded-code construct checking the count as the *if*.

❹ Finally, once matching has completed, we check the count to be sure it's zero. If it's not, there weren't enough close parentheses to balance the opens, so we should fail:

```
(?(?{ $OpenParens != 0 })(?!))
```

Adding these items to the skeleton expression gives us:

```
my $NestedGuts = qr{
  (?{ local $OpenParens = 0 }) # ❶ Counts the number of nested opens waiting to close.
  (?> # atomic-grouping for efficiency
     (?:
        # Stuff not parenthesis
          [^()]+
        # ❷ An opening parenthesis
        |  \(    (?{ $OpenParens++ })
        # ❸ Allow a closing parenthesis, if we're expecting any
        |  \)  (?(?{ $OpenParens != 0 }) (?{ $OpenParens-- }) | (?!) )
     )*
  )
  (?(?{ $OpenParens != 0 })(?!)) # ❹ If there are any open parens left, don't finish
}x;
```

This can now be used just like `$LevelN` on page 330.

The `local` is used as a precaution to isolate this regex's use of `$OpenParens` from any other use the global variable might have within the program. Unlike `local`'s use in the previous section, it's not needed for backtracking protection because the atomic grouping in the regex ensures that once an alternative has been matched, it can't ever be "unmatched." In this case, the atomic grouping is used for both efficiency and to absolutely ensure that the text matched near one of the embedded-code constructs can't be unmatched by backtracking (which would break the sync between the value of `$OpenParens` and the number of parentheses actually matched).

Overloading Regex Literals

You can pre-process the literal parts of a regex literal in any way you like with *overloading*. The next sections show examples.

Adding start- and end-of-word metacharacters

Perl doesn't support ⌈\<⌋ and ⌈\>⌋ as start- and end-of-word metacharacters, and that's probably because it's rare that ⌈\b⌋ doesn't suffice. However, if we wish to have them, we can support them ourselves using overloading to replace '\<' and '\>' in a regex by ⌈(?<!\w)(?=\w)⌋ and ⌈(?<=\w)(?!\w)⌋, respectively.

First, we'll create a function, say, `MungeRegexLiteral`, that does the desired preprocessing:

```
sub MungeRegexLiteral($)
{
   my ($RegexLiteral) = @_; # Argument is a string
   $RegexLiteral =~ s/\\</(?<!\\w)(?=\\w)/g; # Mimic \< as start-of-word boundary
   $RegexLiteral =~ s/\\>/(?<=\\w)(?!\\w)/g; # Mimic \> as end-of-word boundary
   return $RegexLiteral; # Return possibly-modified string
}
```

When this function is passed a string like '`…\<…`', it converts it and returns the string '`…(?<!\w)(?=\w)…`'. Remember, because the replacement part of a substitution is like a double-quoted string, it needs '`\\w`' to get '`\w`' into the value.

Now, to install this so that it gets called automatically on each literal part of a regex literal, we put it into a file, say *MyRegexStuff.pm*, with the Perl mechanics for overloading:

```
package MyRegexStuff; # Best to call the package something unique
use strict;   # Good practice to always use this
use warnings; # Good practice to always use this
use overload; # Allows us to invoke Perl's overloading mechanism
# Have our regex handler installed when we're use'd . . . .
sub import { overload::constant qr => \&MungeRegexLiteral }

sub MungeRegexLiteral($)
{
   my ($RegexLiteral) = @_; # Argument is a string
   $RegexLiteral =~ s/\\</(?<!\\w)(?=\\w)/g; # Mimic \< as start-of-word boundary
   $RegexLiteral =~ s/\\>/(?<=\\w)(?!\\w)/g; # Mimic \> as end-of-word boundary
   return $RegexLiteral; # Return possibly-modified string
}

1; # Standard idiom so that a 'use' of this file returns something true
```

If we place *MyRegexStuff.pm* in the Perl library path (see PERLLIB in the Perl documentation), we can then invoke it from Perl script files in which we want the new features made available. For testing, though, we can just leave it in the same directory as the test script, invoking it with:

```
use lib '.';       # Look for library files in the current directory
use MyRegexStuff; # We now have our new functionality available!
   ⋮
$text =~ s/\s+\</ /g; # Normalize any type of whitespace before a word to a single space
   ⋮
```

We must `use MyRegexStuff` in any file in which we want this added support for regex literals, but the hard work of building *MyRegexStuff.pm* need be done only once. (The new support isn't available in *MyRegexStuff.pm* itself because it doesn't **`use MyRegexStuff`** — something you wouldn't want to do.)

Adding support for possessive quantifiers

Let's extend *MyRegexStuff.pm* to add support for possessive quantifiers like ⌜x++⌟ (☞ 140). Possessive quantifiers work like normal greedy quantifiers, except they never give up (never "unmatch") what they've matched. They can be mimicked with atomic grouping by simply removing the final '+' and wrapping everything in atomic quotes, e.g., by changing ⌜*regex*++⌟ to ⌜(?> *regex*+)⌟ (☞ 173).

The *regex* part can be a parenthesized expression, a metasequence like ⌜\w⌟ or ⌜\x{1234}⌟, or even just a normal character. Handling all possible cases is difficult, so to keep the example simple for the moment, let's concentrate on ?+, *+, or ++ quantifying only a parenthesized expression. Using $LevelN from page 330, we can add

```
$RegexLiteral =~ s/(  \( $LevelN \)[*+?] )\+/(?>$1)/gx;
```

to the MungeRegexLiteral function.

That's it. Now, with it part of our overloaded package, we can use a regex literal with possessive quantifiers, like this example from page 198:

```
$text =~ s/"(\\.|[^"])*+"//; # Remove double-quoted strings
```

Extending this beyond just parenthesized expressions is tricky because of the variety of things that can be in a regular expression. Here's one attempt:

```
$RegexLiteral =~ s{
  (
    # Match something that can be quantified . . .
    (?:  \\[\\abCdDefnrsStwWX] # \n, \w, etc.
      |  \\c. # \cA
      |  \\x[\da-fA-F]{1,2}    # \xFF
      |  \\x\{[\da-fA-F]*\}    # \x{1234}
      |  \\[pP]\{[^{}]+\}      # \p{Letter}
      |  \[\]?[^]]+\]          # "poor man's" class
      |  \\\W                  # \*
      |  \( $LevelN \)         # (…)
      |  [^()*+?\\]            # almost anything else
    )
    #  . . . and is quantified . . .
    (?: [*+?] | \{\d+(?:,\d*)?\} )
  )
  \+   #  . . . and has an extra '+' after the quantifier.
}{(?>$1)}gx;
```

The general form of this regex is the same as before: match something quantified possessively, remove the '+', and wrap the result in ⌜(?>…)⌟. It's only a half-hearted attempt to recognize the complex syntax of Perl regular expressions. The part to match a class is particularly needy, in that it doesn't recognize escapes within the class. Even worse, the basic approach is flawed because it doesn't understand every aspect of Perl regular expressions. For example, if faced with '\(blah\)++', it doesn't properly ignore the opening literal parenthesis, so it thinks the ⌜++⌟ is applied to more than just ⌜\)⌟.

These problems can be overcome with great effort, perhaps using a technique that carefully walks through the regex from start to finish (similar to the approach shown in the sidebar on page 130). I'd like to enhance the part that matches a character class, but in the end, I don't feel it's worth it to address the other issues, for two reasons. The first is that the situations in which it doesn't already work well are fairly contrived, so just fixing the character class part is probably enough to make it acceptable in practice. But in the end, Perl's regex overloading currently has a fatal flaw, discussed in the next section, which renders it much less useful than it might otherwise be.

Problems with Regex-Literal Overloading

Regex-literal overloading can be extremely powerful, at least in theory, but unfortunately, it's not very useful in practice. The problem is that it applies to only the *literal* part of a regex literal, and not the parts that are interpolated. For example, with the code **`m/($MyStuff)*+/`** our `MungeRegexLiteral` function is called twice, once with the literal part of the regex before the variable interpolation ("`(`"), and once with the part after ("`)*+`"). (It's never even given the contents of `$MyStuff`.) Since our function requires both parts at the same time, variable interpolation effectively disables it.

This is less of an issue with the support for `\<` and `\>` we added earlier, since they're not likely to be broken up by variable interpolation. But since overloading doesn't affect the contents of an interpolated variable, a string or regex object containing '`\<`' or '`\>`' would not be processed by overloading. Also, as the previous section touched on, when a regex literal is processed by overloading, it's not easy to be complete and accurate every time. Even something as simple as our support for `\>` gets in the way when given '`\\>`', ostensibly to match a '`\`' followed by '`>`'.

Another problem is that there's no way for the overload processing to know about the modifiers that the regex was applied with. In particular, it may be crucial to know whether `/x` was specified, but there's currently no way to know that.

Finally, be warned that using overloading disables the ability to include characters by their Unicode name (`\N{`*name*`}` ☞ 290).

Mimicking Named Capture

Despite the shortcomings of overloading, I think it's instructive to see a complex example bringing together many special constructs. Perl doesn't offer named capture (☞ 137), but it can be mimicked with capturing parentheses and the `$^N` variable (☞ 301), which references the text matched by the most-recently-closed set of capturing parentheses. (I put on the hat of a Perl developer and added `$^N` support to Perl expressly to allow named-capture to be mimicked.)

As a simple example, consider:

```
⌈href\s*=\s*($HttpUrl)(?{ $url = $^N })⌋
```

This uses the `$HttpUrl` regex object developed on page 303. The underlined part is an embedded-code construct that saves the text matched by `$HttpUrl` to the variable `$url`. In this simple situation, it seems overkill to use `$^N` instead of `$1`, or to even use the embedded-code construct in the first place, since it seems so easy to just use `$1` after the match. But consider encapsulating part of that into a regex object, and then using it multiple times:

```
my $SaveUrl = qr{
    ($HttpUrl)          # Match an HTTP URL . . .
    (?{ $url = $^N })   #  . . . and save to $url
}x;

$text =~ m{
   http \s*=\s* ($SaveUrl)
 | src  \s*=\s* ($SaveUrl)
}xi;
```

Regardless of which matches, `$url` is set to the URL that matched. Again, in this particular use, you could use other means (such as the `$+` variable ☞ 301), but as `$SaveUrl` is used in more complex situations, the other solutions become more difficult to maintain, so saving to a named variable can be much more convenient.

One problem with this example is that values written to `$url` are not "unwritten" when the construct that wrote to them is unmatched via backtracking. So, we need to modify a localized temporary variable during the initial match, writing to the "real" variable only after an overall match has been confirmed, just as we did in the example on page 338.

The listing on the next page shows one way to solve this. From the user's point of view, after using ⌈`(?<Num>\d+)`⌋, the number matched by ⌈`\d+`⌋ is available in the global hash `%^N`, as `$^N{Num}`. Although future versions of Perl could decide to turn `%^N` into a special system variable of some sort, it's not currently special, so we're free to use it.

I could have chosen a name like `%NamedCapture`, but instead chose `%^N` for a few reasons. One is that it's similar to `$^N`. Another is that it's not required to be predeclared with `our` when used under `use strict`. Finally, it's my hope that Perl will eventually add named capture natively, and I think adding it via `%^N` would be a fine idea. If that happens, `%^N` would likely be automatically dynamically scoped like the rest of the regex-related special variables (☞ 299). But as of now, it's a normal global variable, so is not dynamically scoped automatically.

Again, even this more-involved approach suffers from the same problems as anything using regex-literal overloading, such as an incompatibility with interpolated variables.

Mimicking Named Capture

```
package MyRegexStuff;
use strict;
use warnings;
use overload;
sub import { overload::constant('qr' => \&MungeRegexLiteral) }

my $NestedStuffRegex; # This should be predeclared, because it's used in its own definition.
$NestedStuffRegex = qr{
 (?>
   (?:  # Stuff not parens, not '#', and not an escape . . .
        [^()\#\\]+
        # Escaped stuff . . .
      | (?s: \\. )
        # Regex comment . . .
      | \#.*\n
        # Matching parens, with more nested stuff inside . . .
      |  \(  (??{ $NestedStuffRegex })   \)
   )*
 )
}x;

sub SimpleConvert($); # This must be predeclared, as it's used recursively
sub SimpleConvert($)
{
  my $re = shift;  # Regex to mangle
  $re =~ s{
      \(\? #  "(?"
        <  ( (?>\w+) ) >       #     < $1 > $1 is an identifier
        ( $NestedStuffRegex ) #     $2 - possibly-nested stuff
      \)                       #  ")"
  }{
    my $id   = $1;
    my $guts = SimpleConvert($2);
    # We change
    #    (?<id>guts)
    # to
    #    (?: (guts)  # match the guts
    #        (?{
    #           local($^N{$id}) = $guts # Save to a localized element of %^T
    #         })
    #     )
    "(?:($guts)(?{ local(\$^T{'$id'}) = \$^N }))"
  }xeog;
  return $re;  # Return mangled regex
}

sub MungeRegexLiteral($)
{
  my ($RegexLiteral) = @_; # Argument is a string
  # print "BEFORE: $RegexLiteral\n"; # Uncomment this for debugging
  my $new = SimpleConvert($RegexLiteral);
  if ($new ne $RegexLiteral)
  {
     my $before = q/(?{ local(%^T) = () })/; # Localize temporary hash
     my $after  = q/(?{ %^N = %^T       })/; # Copy temp to "real" hash
     $RegexLiteral = "$before(?:$new)$after";
  }
  # print "AFTER:  $RegexLiteral\n"; # Uncomment this for debugging
  return $RegexLiteral;
}

1;
```

Perl Efficiency Issues

For the most part, efficiency with Perl regular expressions is achieved in the same way as with any tool that uses a Traditional NFA. Use the techniques discussed in Chapter 6—the internal optimizations, the unrolling methods, the "Think" section—all apply to Perl.

There are, of course, Perl-specific issues as well, and in this section, we'll look at the following topics:

- **There's More Than One Way To Do It** Perl is a toolbox offering many approaches to a solution. Knowing which problems are nails comes with understanding *The Perl Way*, and knowing which hammer to use for any particular nail goes a long way toward making more efficient and more understandable programs. Sometimes efficiency and understandability seem to be mutually exclusive, but a better understanding allows you to make better choices.
- **Regex Compilation, `qr/···/`, the `/o` Modifier, and Efficiency** The interpolation and compilation of regex operands are fertile ground for saving time. The `/o` modifier, which I haven't discussed much yet, along with regex objects (`qr/···/`), gives you some control over when the costly re-compilation takes place.
- **The `$&` Penalty** The three match side effect variables, `$`, `$&`, and `$'`, can be convenient, but there's a hidden efficiency *gotcha* waiting in store for any script that uses them, even once, anywhere. Heck, you don't even have to *use* them—the entire script is penalized if one of these variables even *appears* in the script.
- **The Study Function** Since ages past, Perl has provided the `study(···)` function. Using it supposedly makes regexes faster, but it seems that no one really understands if it does, or why. We'll see whether we can figure it out.
- **Benchmarking** When it comes down to it, the fastest program is the one that finishes first. (You can quote me on that.) Whether a small routine, a major function, or a whole program working with live data, benchmarking is the final word on speed. Benchmarking is easy and painless with Perl, although there are various ways to go about it. I'll show you the way I do it, a simple method that has served me well for the hundreds of benchmarks I've done while preparing this book.
- **Perl's Regex Debugging** Perl's regex-debug flag can tell you about some of the optimizations the regex engine and transmission do, or don't do, with your regexes. We'll look at how to do this and see what secrets Perl gives up.

"There's More Than One Way to Do It"

There are often many ways to go about solving any particular problem, so there's no substitute for really knowing all that Perl has to offer when balancing efficiency and readability. Let's look at the simple problem of padding an IP address like '`18.181.0.24`' such that each of the four parts becomes exactly three digits: '`018.181.000.024`'. One simple and readable solution is:

```
$ip = sprintf("%03d.%03d.%03d.%03d", split(/\./, $ip));
```

This is a fine solution, but there are certainly other ways to do the job. In the interest of comparison, Table 7-6 examines various ways to achieve the same goal, and their relative efficiency (they're listed from the most efficient to the least). This example's goal is simple and not very interesting in and of itself, yet it represents a common text-handling task, so I encourage you to spend some time understanding the various approaches. You may even see some Perl techniques that are new to you.

Each approach produces the same result when given a correct IP address, but fails in different ways if given something else. If there is any chance that the data will be malformed, you'll need more care than any of these solutions provide. That aside, the practical differences lie in efficiency and readability. As for readability, #1 and #13 seem the most straightforward (although it's interesting to see the wide gap in efficiency). Also straightforward are #3 and #4 (similar to #1) and #8 (similar to #13). The rest all suffer from varying degrees of crypticness.

So, what about efficiency? Why are some less efficient than others? It's the interactions among how an NFA works (Chapter 4), Perl's many regex optimizations (Chapter 6), and the speed of other Perl constructs (such as `sprintf`, and the mechanics of the substitution operator). The substitution operator's `/e` modifier, while indispensable at times, does seem to be mostly at the bottom of the list.

It's interesting to compare two pairs, #3/#4 and #8/#14. The two regexes of each pair differ only in their use of parentheses — the one without the parentheses is just a bit faster than the one with. But #8's use of `$&` as a way to avoid parentheses comes at a high cost not shown by these benchmarks (☞ 355).

Regex Compilation, the /o Modifier, `qr/···/`, and Efficiency

An important aspect of Perl's regex-related efficiency relates to the setup work Perl must do behind the scenes when program execution reaches a regex operator, before actually applying the regular expression. The precise setup depends on the

Table 7-6: A Few Ways to Pad an IP Address

Rank	Time	Approach
1.	1.0×	`$ip = sprintf("%03d.%03d.%03d.%03d", split(m/\./, $ip));`

2.	1.3×	`substr($ip,  0, 0) = '0' if substr($ip,  1, 1) eq '.';`
		`substr($ip,  0, 0) = '0' if substr($ip,  2, 1) eq '.';`
		`substr($ip,  4, 0) = '0' if substr($ip,  5, 1) eq '.';`
		`substr($ip,  4, 0) = '0' if substr($ip,  6, 1) eq '.';`
		`substr($ip,  8, 0) = '0' if substr($ip,  9, 1) eq '.';`
		`substr($ip,  8, 0) = '0' if substr($ip, 10, 1) eq '.';`
		`substr($ip, 12, 0) = '0' while length($ip) < 15;`

3.	1.6×	`$ip = sprintf("%03d.%03d.%03d.%03d", $ip =~ m/\d+/g);`
4.	1.8×	`$ip = sprintf("%03d.%03d.%03d.%03d", $ip =~ m/(\d+)/g);`
5.	1.8×	`$ip = sprintf("%03d.%03d.%03d.%03d",` `$ip =~ m/^(\d+)\.(\d+)\.(\d+)\.(\d+)$/);`
6.	2.3×	`$ip =~ s/\b(?=\d\b)/00/g;` `$ip =~ s/\b(?=\d\d\b)/0/g;`
7.	3.0×	`$ip =~ s/\b(\d(\d?)\b)/$2 eq '' ? "00$1" : "0$1"/eg;`
8.	3.3×	`$ip =~ s/\d+/sprintf("%03d", $&)/eg;`
9.	3.4×	`$ip =~ s/(?:(?<=\.)\|^)(?=\d\b)/00/g;` `$ip =~ s/(?:(?<=\.)|^)(?=\d\d\b)/0/g;`
10.	3.4×	`$ip =~ s/\b(\d\d?\b)/'0' x (3-length($1)) . $1/eg;`
11.	3.4×	`$ip =~ s/\b(\d\b)/00$1/g;` `$ip =~ s/\b(\d\d\b)/0$1/g;`
12.	3.4×	`$ip =~ s/\b(\d\d?\b)/sprintf("%03d", $1)/eg;`
13.	3.5×	`$ip =~ s/\b(\d{1,2}\b)/sprintf("%03d", $1)/eg;`
14.	3.5×	`$ip =~ s/(\d+)/sprintf("%03d", $1)/eg;`
15.	3.6×	`$ip =~ s/\b(\d\d?(?!\d))/sprintf("%03d", $1)/eg;`
16.	4.0×	`$ip =~ s/(?:(?<=\.)|^)(\d\d?(?!\d))/sprintf("%03d", $1)/eg;`

type of regex operand. In the most common situation, the regex operand is a regex literal, as with `m/…/` or `s/…/…/` or `qr/…/`. For these, Perl has to do a few different things behind the scenes, each taking some time we'd like to avoid, if possible. First, let's look at what needs to be done, and then at ways we might avoid it.

The internal mechanics of preparing a regex

The behind-the-scenes work done to prepare a regex operand is discussed generally in Chapter 6 (☞ 241), but Perl has its unique twists.

Perl's pre-processing of regex operands happens in two general phases.

1. **Regex-literal processing** If the operand is a regex literal, it's processed as described in "How Regex Literals Are Parsed" (☞ 292). One of the benefits provided by this stage is variable interpolation.
2. **Regex Compilation** The regex is inspected, and if valid, compiled into an internal form appropriate for its actual application by the regex engine. (If invalid, the error is reported to the user.)

Once Perl has a compiled regex in hand, it can actually apply it to the target string, as per Chapters 4-6.

All that pre-processing doesn't necessarily need be done every time each regex operator is used. It must always be done the *first* time a regex literal is used in a program, but if execution reaches the same regex literal more than once (such as in a loop, or in a function that's called more than once), Perl can sometimes re-use some of the previously-done work. The next sections show when and how Perl might do this, and additional techniques available to the programmer to further increase efficiency.

Perl steps to reduce regex compilation

In the next sections, we'll look at two ways in which Perl avoids some of the pre-processing associated with regex literals: *unconditional caching* and *on-demand recompilation*.

Unconditional caching

If a regex literal has no variable interpolation, Perl knows that the regex can't change from use to use, so after the regex is compiled once, that compiled form is saved ("cached") for use whenever execution again reaches the same code. The regex is examined and compiled just once, no matter how often it's used during the program's execution. Most regular expressions shown in this book have no variable interpolation, and so are perfectly efficient in this respect.

Variables within embedded code and dynamic regex constructs don't count, as they're not *interpolated* into the value of the regex, but rather part of the unchanging code the regex executes. When `my` variables are referenced from within embedded code, there may be times that you wish it were interpreted every time: see the warning on page 338.

Just to be clear, caching lasts only as long as the program executes — nothing is cached from one run to the next.

On-demand recompilation

Not all regex operands can be cached directly. Consider this snippet:

```
my $today = (qw<Sun Mon Tue Wed Thu Fri Sat>)[(localtime)[6]];
# $today now holds the day ("Mon", "Tue", etc., as appropriate)

while (<LOGFILE>) {
    if (m/^$today:/i) {
        ⋮
```

The regex in `m/^$today:/` requires interpolation, but the way it's used in the loop ensures that the result of that interpolation will be the same every time. It would be inefficient to recompile the same thing over and over each time through the loop, so Perl automatically does a simple string check, comparing the result of the interpolation against the result the last time through. If they're the same, the cached regex that was used the previous time is used again this time, eliminating the need to recompile. But if the result of the interpolation turns out to be different, the regex is recompiled. So, for the price of having to redo the interpolation and check the result with the cached value, the relatively expensive compile is avoided whenever possible.

How much do these features actually save? Quite a lot. As an example, I benchmarked the cost of pre-processing three forms of the `$HttpUrl` example from page 303 (using the extended `$HostnameRegex`). I designed the benchmarks to show the overhead of regex pre-processing (the interpolation, string check, compilation, and other background tasks), not the actual application of the regex, which is the same regardless of how you get there.

The results are pretty interesting. I ran a version that has no interpolation (the entire regex manually spelled out within `m/⋯/`), and used that as the basis of comparison. The interpolation and check, if the regex doesn't change each time, takes about 25× longer. The full pre-processing (which adds the recompilation of the regex each time) takes about 1,000× longer! Wow.

Just to put these numbers into context, realize that even the full pre-processing, despite being over 1,000× slower than the static regex literal pre-processing, still takes only about 0.00026 seconds on my system. (It benchmarked at a rate of about 3,846 per second; on the other hand, the static regex literal's pre-processing benchmarked at a rate of about 3.7 million per second.) Still, the savings of not having to do the interpolation are impressive, and the savings of not having to recompile are down right fantastic. In the next sections, we'll look at how you can take action to enjoy these savings in even more cases.

The "compile once" /o modifier

Put simply, if you use the `/o` modifier with a regex literal operand, the regex literal will be inspected and compiled just once, regardless of whether it uses interpolation. If there's no interpolation, adding `/o` doesn't buy you anything because expressions without interpolation are always cached automatically. If there *is* interpolation, the first time execution arrives at the regex literal, the normal full pre-processing happens, but because of `/o`, the internal form is cached. If execution comes back again to the same regex operator, that cached form is used directly.

Here's the example from the previous page, with the addition of `/o`:

```
my $today = (qw<Sun Mon Tue Wed Thu Fri Sat>)[(localtime)[6]];

while (<LOGFILE>) {
    if (m/^$today:/io) {
        ⋮
```

This is now much more efficient because the regex ignores `$today` on all but the first iteration through the loop. Not having to interpolate or otherwise pre-process and compile the regex every time represents a real savings that Perl couldn't do for us automatically because of the variable interpolation: `$today` *might* change, so Perl must play it safe and reinspect it each time. By using `/o`, we tell Perl to "lock in" the regex after the regex literal is first pre-processed and compiled. It's safe to do this when we know that the variables interpolated into a regex literal won't change, or when we don't want Perl to use the new values even if they do change.

Potential "gotchas" of /o

There's an important "gotcha" to watch out for with `/o`. Consider putting our example into a function:

```
sub CheckLogfileForToday()
{
  my $today = (qw<Sun Mon Tue Wed Thu Fri Sat>)[(localtime)[6]];

  while (<LOGFILE>) {
      if (m/^$today:/io) { #dangerous -- has a gotcha
          ⋮
      }
  }
}
```

Remember, `/o` indicates that the regex operand should be compiled *once*. The first time `CheckLogfileForToday()` is called, a regex operand representing the current day is locked in. If the function is called again some time later, even though `$today` may change, it will not be not inspected again; the original locked-in regex is used every time for the duration of execution.

This is a major shortcoming, but as we'll see in the next section, regex objects provide a best-of-both-worlds way around it.

Using regex objects for efficiency

All the discussion of pre-processing we've seen so far applies to regex *literals*. The goal has been to end up with a compiled regex with as little work as possible. Another approach to the same end is to use a regex *object*, which is basically a ready-to-use compiled regex encapsulated into a variable. They're created with the `qr/···/` operator (☞ 303).

Here's a version of our example using a regex object:

```
sub CheckLogfileForToday()
{
  my $today = (qw<Sun Mon Tue Wed Thu Fri Sat>)[(localtime)[6]];
  my $RegexObj = qr/^$today:/i; # compiles once per function call
  while (<LOGFILE>) {
      if ($_ =~ $RegexObj) {
            ⋮
      }
  }
}
```

Here, a new regex object is created each time the function is called, but it is then used directly for each line of the log file. When a regex object is used as an operand, it undergoes none of the pre-processing discussed throughout this section. The pre-processing is done when the regex object is *created*, not when it's later *used*. You can think of a regex object, then, as a "floating regex cache," a ready-to-use compiled regex that you can apply whenever you like.

This solution has the best of both worlds: it's efficient, since only one regex is compiled during each function call (not with each line in the log file), but, unlike the previous example where `/o` was used inappropriately, this example actually works correctly with multiple calls to `CheckLogfileForToday()`.

Be sure to realize that there are two regex operands in this example. The regex operand of the `qr/···/` is *not* a regex object, but a regex literal supplied to `qr/···/` to *create* a regex object. The object is then used as the regex operand for the `=~` match operator in the loop.

Using m/···/ with regex objects

The use of the regex object,

```
if ($_ =~ $RegexObj) {
```

can also be written as:

```
if (m/$RegexObj/) {
```

This is not a normal regex literal, even though it looks like one. When the only thing in the "regex literal" is a regex object, it's just the same as using a regex object. This is useful for several reasons. One is simply that the `m/···/` notation may be more familiar, and perhaps more comfortable to work with. It also relieves you from explicitly stating the target string `$_`, which makes things look better in conjunction with other operators that use the same default. Finally, it allows you to use the `/g` modifier with regex objects.

Using /o with qr/···/

The `/o` modifier can be used with `qr/···/`, but you'd certainly not want to in this example. Just as when `/o` is used with any of the other regex operators, `qr/···/o` locks in the regex the first time it's used, so if used here, `$RegexObj` would get the same regex object each time the function is called, regardless of the value of `$today`. That would be the same mistake as when we used `m/···/o` on page 352.

Using the default regex for efficiency

The default regex (☞308) feature of regex operators can be used for efficiency, although the need for it has mostly been eliminated with the advent of regex objects. Still, I'll describe it quickly. Consider:

```
sub CheckLogfileForToday()
{
  my $today = (qw<Sun Mon Tue Wed Thu Fri Sat>)[(localtime)[6]];

  # Keep trying until one matches, so the default regex is set.
  "Sun:" =~ m/^$today:/i or
  "Mon:" =~ m/^$today:/i or
  "Tue:" =~ m/^$today:/i or
  "Wed:" =~ m/^$today:/i or
  "Thu:" =~ m/^$today:/i or
  "Fri:" =~ m/^$today:/i or
  "Sat:" =~ m/^$today:/i;

  while (<LOGFILE>) {
      if (m//) { # Now use the default regex
          ⋮
      }
  }
}
```

The key to using the default regex is that a match must be successful for it to be set, which is why this example goes to such trouble to get a match after `$today` has been set. As you can see, it's fairly kludgey, and I wouldn't recommend it.

Understanding the "Pre-Match" Copy

While doing matches and substitutions, Perl sometimes must spend extra time and memory to make a pre-match copy of the target text. As we'll see, sometimes this copy is used in support of important features, but sometimes it's not. When the copy is made but not used, the wasted effort is an inefficiency we'd like to avoid, especially in situations where the target text is very long, or speed particularly important.

In the next sections, we'll look at when and why Perl might make a pre-match copy of the target text, when the copy is actually used, and how we might avoid the copy when efficiency is at a premium.

Pre-match copy supports $1, $&, $', $+, ...

Perl makes a pre-match copy of the original target text of a match or substitution to support `$1`, `$&`, and the other after-match variables that actually hold text (☞ 299). After each match, Perl doesn't actually create each of these variables because many (or all) may never be used by the program. Rather, Perl just files away a copy of the original text, remembers *where* in that original string the various matches happened, and then refers to that if and when `$1` or the like is actually used. This requires less work up-front, which is good, because often, some or all of these after-match variables are not even used. This is a form of "lazy evaluation," and successfully avoids a lot of unneeded work.

Although Perl saves work by not creating `$1` and the like until they're used, it still has to do the work of saving the extra copy of the target text. But why does this really need to be done? Why can't Perl just refer to that original text to begin with? Well, consider:

```
$Subject =~ s/^(?:Re:\s*)+//;
```

After this, `$&` properly refers to the text that was removed from `$Subject`, but since it was *removed* from `$Subject`, Perl can't refer to `$Subject` itself when providing for a subsequent use of `$&`. The same logic applies for something like:

```
if ($Subject =~ m/^SPAM:(.+)/i) {
    $Subject = "-- spam subject removed --";
    $SpamCount{$1}++;
}
```

By the time `$1` is referenced, the original `$Subject` has been erased. Thus, Perl must make an internal pre-match copy.

The pre-match copy is not always needed

In practice, the primary "users" of the pre-match copy are `$1`, `$2`, `$3`, and the like. But what if a regex doesn't even have capturing parentheses? If it doesn't, there's

no need to even worry about $1, so any work needed to support it can be bypassed. So, at least those regexes that don't have capturing parentheses can avoid the costly copy? Not always . . .

The variables $`, $&, and $' are naughty

The three variables $`, $&, and $' aren't related to capturing parentheses. As the text before, of, and after the match, they can potentially apply to *every* match and substitution. Since it's impossible for Perl to tell which match any particular use of one of these variables refers to, Perl must make the pre-match copy *every time*.

It might sound like there's no opportunity to avoid the copy, but Perl is smart enough to realize that if these variables do not appear in the program, *anywhere* (including in any library that might be used) the blind copying to support them is no longer needed. **Thus, ensuring that you don't use $`, $&, and $' allows all matches without capturing parentheses to dispense with the pre-match copy — a handsome optimization!** Having even one $`, $&, or $' anywhere in the program means the optimization is lost. How unsociable! For this reason, I call these three variables "naughty."

How expensive is the pre-match copy?

I ran a simple benchmark, checking m/c/ against each of the 130,000 lines of C that make up the main Perl source. The benchmark noted whether a 'c' appeared on each line, but didn't do anything further, since the goal was to determine the effect of the behind-the-scenes copying. I ran the test two different ways: once where I made sure not to trigger the pre-match copy, and once where I made sure to do so. The only difference, therefore, was in the extra copy overhead.

The run with the pre-match copying consistently took over 40 percent longer than the one without. This represents an "average worst case," so to speak, since the benchmark didn't do any "real work," whose time would reduce the relative relevance of (and perhaps overshadow) the extra overhead.

On the other hand, in true worst-case scenarios, the extra copy might truly be an overwhelming portion of the work. I ran the same test on the same data, but this time as *one huge line* incorporating the more than 3.5 megabytes of data, rather than the 130,000 or so reasonably sized lines. Thus, the relative performance of a single match can be checked. The match without the pre-match copy returned almost immediately, since it was sure to find a 'c' somewhere near the start of the string. Once it did, it was finished. The test *with* the pre-match copy is the same except that it had to make a copy of the huge string first. It took over 7,000 times longer! Knowing the ramifications, therefore, of certain constructs allows you to tweak your code for better efficiency.

Avoiding the pre-match copy

It would be nice if Perl knew the programmer's intentions and made the copy only as necessary. But remember, the copies are not "bad" — Perl's handling of these bookkeeping drudgeries behind the scenes is why we use it and not, say, C or assembly language. Indeed, Perl was first developed in part to free users from the mechanics of bit fiddling so they could concentrate on creating solutions to problems.

Never use naughty variables. Still, it's nice to avoid the extra work if possible. Foremost, of course, is to never use `$``, `$&`, or `$'` *anywhere* in your code. Often, `$&` is easy to eliminate by wrapping the regex with capturing parentheses, and using `$1` instead. For example, rather than using **`s/<\w+>/\L$&\E/g`** to lowercase certain HTML tags, use **`s/(<\w+>)/\L$1\E/g`** instead.

`$`` and `$'` can often be easily mimicked if you still have an unmodified copy of the original target string. After a match against a given *target*, the following shows valid replacements:

Variable	Mimicked with
`$``	`substr(`*target*`, 0, $-[0])`
`$&`	`substr(`*target*`, $-[0], $+[0] - $-[0])`
`$'`	`substr(`*target*`, $+[0])`

Since `@-` and `@+` (☞302) are arrays of *positions* in the original target string, rather than actual *text* in it, they can be safely used without an efficiency penalty.

I've included a substitute for `$&` in there as well. This may be a better alternative to wrapping with capturing parentheses and using `$1`, as it may allow you to eliminate capturing parentheses altogether. Remember, the whole point of avoiding `$&` and friends is to avoid the copy for matches that have no capturing parentheses. If you make changes to your program to eliminate `$&`, but end up adding capturing parentheses to every match, you haven't saved anything.

Don't use naughty modules. Of course, part of not using `$``, `$&`, or `$'` is to not use modules that use them. The core modules that come with Perl do not use them, except for the `English` module. If you wish to use that module, you can have it not apply to these three variables by invoking it as:

```
use English '-no_match_vars';
```

This makes it safe. If you download modules from CPAN or elsewhere, you may wish to check to see if they use the variables. See the sidebar on the next page for a technique to check to see if your program is infected with any of these variables.

How to Check Whether Your Code is Tainted by $&

It's not always easy to notice whether your program is naughty (references `$&`, `$`` ` ``, or `$'`), especially with the use of libraries, but there are several ways to find out. The easiest, if your *perl* binary has been compiled with the `-DDEBUGGING` option, is to use the **`-c`** and **`-Mre=debug`** command-line arguments (☞361) and look toward the end of the output for a line that says either '**`Enabling $` $& $' support`**' or '**`Omitting $` $& $' support`**'. If it's enabled, the code is tainted.

It's possible (but unlikely) that the code could be tainted by the use of a naughty variable within an eval that's not known to Perl until it's executed. One option to catch those as well is to install the `Devel::SawAmpersand` package from CPAN (`http://www.cpan.org`):

```
END {
   require Devel::SawAmpersand;
   if (Devel::SawAmpersand::sawampersand) {
       print "Naughty variable was used!\n";
   }
 }
```

Included with `Devel::SawAmpersand` comes `Devel::FindAmpersand`, a package that purportedly shows you where the offending variable is located. Unfortunately, it doesn't work reliably with the latest versions of Perl. Also, they both have some installation issues, so your mileage may vary. (Check `http://regex.info/` for possible updates.)

Also, it may be interesting to see how you can check for naughtiness by just checking for the performance penalty:

```
use Time::HiRes;
sub CheckNaughtiness()
{
  my $text = 'x' x 10_000; # Create some non-small amount of data.

  # Calculate the overhead of a do-nothing loop.
  my $start = Time::HiRes::time();
  for (my $i = 0; $i < 5_000; $i++)  {  }
  my $overhead = Time::HiRes::time() - $start;

  # Now calculate the time for the same number of simple matches.
  $start = Time::HiRes::time();
  for (my $i = 0; $i < 5_000; $i++)  { $text =~ m/^/  }
  my $delta = Time::HiRes::time() - $start;

  # A differential of 5 is just a heuristic.
  printf "It seems your code is %s (overhead=%.2f, delta=%.2f)\n",
    ($delta > $overhead*5) ? "naughty" : "clean", $overhead, $delta;
}
```

The Study Function

In contrast to optimizing the regex itself, `study(···)` optimizes certain kinds of searches of a *string*. After studying a string, a regex (or multiple regexes) can benefit from the cached knowledge when applied to the string. It's generally used like this:

```
while (<>)
{
    study($_); # Study the default target $_ before doing lots of matches on it
    if (m/regex 1/) { ··· }
    if (m/regex 2/) { ··· }
    if (m/regex 3/) { ··· }
    if (m/regex 4/) { ··· }
}
```

What `study` does is simple, but understanding when it's a benefit can be quite difficult. It has no effect whatsoever on any values or results of a program—the only effects are that Perl uses more memory, and that overall execution time might increase, stay the same, or (here's the goal) decrease.

When a string is studied, Perl takes some time and memory to build a list of places in the string that each character is found. (On most systems, the memory required is four times the size of the string). `study`'s benefit can be realized with each subsequent regex match against the string, but only until the string is modified. Any modification of the string invalidates the `study` list, as does studying a different string.

How helpful it is to have the target string `study`ied is highly dependent on the regex matching against it, and the optimizations that Perl is able to apply. For example, searching for literal text with `m/foo/` can see a huge speedup due to `study` (with large strings, speedups of 10,000× are possible). But, if `/i` is used, that speedup evaporates, as `/i` currently removes the benefit of `study` (as well as some other optimizations).

When not to use study

- Don't use `study` on strings you intend to check only with `/i`, or when all literal text is governed by ⌜`(?i)`⌟ or ⌜`(?i:···)`⌟, as these disable the benefits of `study`.
- Don't use `study` when the target string is short. In such cases, the normal fixed-string cognizance optimization should suffice (☞ 247). How short is "short"? String length is just one part of a large, hard-to-pin-down mix, so when it comes down to it, only benchmarking *your* expressions on *your* data will tell you if `study` is a benefit. But for what it's worth, I generally don't even consider `study` unless the strings are at least several kilobytes long.

- Don't use `study` when you plan only a few matches against the target string before it's modified, or before you `study` a different string. An overall speedup is more likely if the time spent to `study` a string is amortized over many matches. With just a few matches, the time spent building the `study` list can overshadow any savings.
- Use `study` only on strings that you intend to search with regular expressions having "exposed" literal text (☞ 255). Without a known character that must appear in any match, `study` is useless. (Along these lines, one might think that `study` would benefit the `index` function, but it doesn't seem to.)

When study can help

`study` is best used when you have a large string you intend to match many times before the string is modified. A good example is a filter I use in the preparation of this book. I write in a home-grown markup that the filter converts to SGML (which is then converted to *troff*, which is then converted to PostScript). Within the filter, an entire chapter eventually ends up within one huge string (for instance, this chapter is about 475KB). Before exiting, I apply a bevy of checks to guard against mistaken markup leaking through. These checks don't modify the string, and they often look for fixed strings, so they're what `study` thrives on.

Benchmarking

If you really care about efficiency, it may be best to try benchmarking. Perl comes standard with the `Benchmark` module, which has fine documentation ("`perldoc Benchmark`"). Perhaps more out of habit than anything else, I tend to write my benchmarks from scratch. After

```
use Time::HiRes 'time';
```

I wrap what I want to test in something simple like:

```
my $start = time;
  ⋮
my $delta = time - $start;
printf "took %.1f seconds\n", $delta;
```

Important issues with benchmarking include making sure to benchmark enough work to show meaningful times, and to benchmark as much of the work you want to measure while benchmarking as little of the work you don't. This is discussed in more detail in Chapter 6 (☞ 232). It might take some time to get used to benchmarking in a reasonable way, but the results can be quite enlightening and rewarding.

Regex Debugging Information

Perl carries out a phenomenal number of optimizations to try to arrive at a match result quickly; some of the less esoteric ones are listed in Chapter 6's "Common Optimizations" (☞ 239), but there are many more. Most optimizations apply to only very specific cases, so any particular regex benefits from only some (or none) of them.

Perl has debugging modes that tell you about some of the optimizations. When a regex is first compiled, Perl figures out which optimizations go with the regex, and the debugging mode reports on some of them. The debugging modes can also tell you a lot about how the engine actually applies that expression. A detailed analysis of this debugging information is beyond the scope of even this book, but I'll provide a short introduction here.

You can turn on the debugging information by putting **`use re 'debug';`** in your code, and you can turn it back off with **`no re 'debug';`**. (We've seen this `use re` pragma before, with different arguments, to allow embedded code in interpolated variables ☞ 337.)

Alternatively, if you want to turn it on for the entire script, you can use the `-Mre=debug` command-line argument. This is particularly useful just for inspecting how a single regex is compiled. Here's an example (edited to remove some lines that are not of interest):

```
❶ % perl -cw -Mre=debug -e 'm/^Subject: (.*)/'
❷ Compiling REx '^Subject: (.*)'
❸ rarest char j at 3
❹    1: BOL(2)
❺    2: EXACT <Subject: >(6)
            ⋮
❻   12: END(0)
❼ anchored 'Subject: ' at 0 (checking anchored) anchored(BOL) minlen 9
❽ Omitting $` $& $' support.
```

At ❶, I invoke *perl* at my shell prompt, using the command-line flags **`-c`** (which means to check the script, but don't actually execute it), **`-w`** (issue warnings about things Perl thinks are dubious — always used as a matter of principle), and **`-Mre=debug`** to turn on regex debugging. The **`-e`** flag means that the following argument, '`m/^Subject:␣(.*)/`', is actually a mini Perl program to be run or checked.

Line ❸ reports the "rarest" character (the least common, as far as Perl guesses) from among those in the longest fixed substring part of the regex. Perl uses this for some optimizations (such as pre-check of required character/substring ☞ 244).

Lines ❹ through ❻ represents Perl's compiled form of the regex. For the most part, we won't be concerned much about it here. However, in even a casual look, line ❺ sticks out as understandable.

Line ❼ is where most of the action is. Some of the information that might be shown here includes:

`anchored '`*string*`' at` *offset*
: Indicates that any match must have the given *string*, starting *offset* characters from the start of the match. If '`$`' is shown immediately after '*string*', the *string* also ends the match.

`floating '`*string*`' at` *from*`..`*to*
: Indicates that any match must have the given *string*, but that it could start anywhere from *from* characters into the match, to *to* characters. If '`$`' is shown immediately after '*string*', the *string* also ends the match.

`stclass '`*list*`'`
: Shows the list of characters with which a match can begin.

`anchored(MBOL)`, `anchored(BOL)`, `anchored(SBOL)`
: The regex leads with ⌈`^`⌋. The `MBOL` version appears when the `/m` modifier is used, while `BOL` and `SBOL` appear when it's is not used. (The difference between `BOL` and `SBOL` is not relevant for modern Perl. `SBOL` relates to the regex-related `$*` variable, which has long been deprecated.)

`anchored(GPOS)`
: The regex leads with ⌈`\G`⌋.

`implicit`
: The `anchored(MBOL)` is an implicit one added by Perl because the regex begins with ⌈`.*`⌋.

`minlen` *length*
: Any match is at least *length* characters long.

`with eval`
: The regex has ⌈`(?{…})`⌋ or ⌈`(??{…})`⌋.

Line ❽ is not related to any particular regex, and appears only if your *perl* binary has been compiled with `-DDEBUGGING` turned on. With it, after loading the whole program, Perl reports if support for `$&` and friends has been enabled (☞ 356).

Run-time debugging information

We've already seen how we can use embedded code to get information about how a match progresses (☞ 331), but Perl's regex debugging can show much more. If you omit the `-c` compile-only option, Perl displays quite a lot of information detailing just how each match progresses.

If you see "Match rejected by optimizer," it means that one of the optimizations enabled the transmission to realize that the regex could never match the target text, and so the application is bypassed altogether. Here's an example:

```
% perl -w -Mre=debug -e '"this is a test" =~ m/^Subject:/;'
    ⋮
Did not find anchored substr 'Subject:'...
Match rejected by optimizer
```

When debugging is turned on, you'll see the debugging information for any regular expressions that are used, not necessarily just your own. For example

```
% perl -w -Mre=debug -e 'use warnings'
. . . lots of debugging information . . .
            ⋮
```

does nothing more than load the `warnings` module, but because that module has regular expressions, you see a lot of debugging information.

Other ways to invoke debugging messages

I've mentioned that you can use "`use re 'debug';`" or `-Mre=debug` to turn on regex debug information. However, if you use `debugcolor` instead of `debug` with either of these, and if you are using a terminal that understands ANSI terminal control escape sequences, the information is shown with highlighting that makes the output easier to read.

Another option is that if your perl binary has been compiled with extra debugging support turned on, you can use the `-Dr` command-line flag as a shorthand for `-Mre=debug`.

Final Comments

I'm sure it's obvious that I'm quite enamored with Perl's regular expressions, and as I noted at the start of the chapter, it's with good reason. Larry Wall, Perl's creator, apparently let himself be ruled by common sense and the Mother of Invention. Yes, the implementation has its warts, but I still allow myself to enjoy the delicious richness of the regex language and the integration with the rest of Perl.

However, I'm not a blind fanatic—Perl does not offer features that I wish for. Since several of the features I pined for in the first edition of this book were eventually added, I'll go ahead and wish for more here. The most glaring omission offered by other implementations is named capture (☞ 137). This chapter offers a way to mimic them, but with severe restrictions; it would be much nicer if they were built in. Class set operations (☞ 123) would also be very nice to have, even though with some effort, they can already be mimicked with lookaround (☞ 124).

Then there are possessive quantifiers (☞ 140). Perl has atomic grouping, which offers more overall functionality, but still, possessive quantifiers offer a clearer, more elegant solution in some situations. So, I'd like both notations. In fact, I'd also like two related constructs that no flavor currently offers. One is a simple "cut" operator, say ⌈`\v`⌋, which would immediately flush any saved states that currently exist (with this, ⌈`x+\v`⌋ would be the same as the possessive ⌈`x++`⌋ or the atomic grouping ⌈`(?>x+)`⌋). The other related construct I'd like would take the additional step of prohibiting any further bump-alongs by the transmission. It would mean "either a match is found from the current path I'm on, or no match will be allowed, period." Perhaps ⌈`\V`⌋ would be a good notation for that.

Somewhat related to my idea for ⌈`\V`⌋, I think that it would be useful to somehow have general hooks into the transmission. This would make it easier to do what we did on page 335.

Finally, as I mentioned on page 337, I think it would be nice to have more control over when embedded code can be interpolated into a regex.

Perl is not the ideal regex-wielding language, but it is very close, and is always getting better. In fact, as this book is going to print, Larry Wall is forging ahead on the design of Perl 6, including a recently-released paper describing his radical new ideas for the future of regular expressions. It will still be some while before Perl 6 is a reality, but the future certainly looks exciting.