Regular Expressions

Regular expressions are the most obvious very high-level feature of Perl. A single pattern match in Perl—even a simple one—can perform the work of many lines in a different language. Pattern matches, especially when combined with Perl's handling of strings and lists, provide capabilities that are very difficult to mimic in other programming languages.

The power of regular expressions is one thing. Making use of it is another. Getting the full benefit from regular expressions in Perl requires both experience and understanding. Becoming fluent in regular expressions may seem to be a difficult task, but I commend it to you. Once you have mastered regular expressions in Perl, your programs will be faster, shorter, and easier to write. In other words, more *effective*—which is why you are reading this book, right?

This section discusses many commonly-encountered issues relating to regular expressions. It is not a reference, however. For a complete description of regular expressions and Perl, see the Perl man pages and/or the Camel book. For an illuminating and extremely thorough discussion of regular expressions that reaches far beyond Perl, see Jeffrey Friedl's excellent *Mastering Regular Expressions*, the so-called "Hip Owls book."

Item 15: Know the precedence of regular expression operators.

The "expression" in "regular expression" is there because regular expressions are constructed and parsed using grammatical rules that are similar to those used for arithmetic expressions. Although regular expressions serve a greatly different purpose, understanding the similarities between them will help you write better regular expressions, and hence better Perl.

Regular expressions in Perl are made up of *atoms*. Atoms are connected by *operators* like repetition, sequence, and alternation. Most regular expression atoms are single-character matches. For example:

`a` *Matches the letter* a.

`\$` *Matches the character* $*—backslash escapes metacharacters.*

`\n` *Matches newline.*

`[a-z]` *Matches a lowercase letter.*

`.` *Matches any character except* \n.

`\1` *Matches contents of first memory—arbitrary length.*

There are also special "zero-width" atoms. For example:

`\b` *Word boundary—transition from* \w *to* \W.

`^` *Matches start of a string.*

`\Z` *Matches end of a string or before newline at end.*

Atoms are modified and/or joined together by regular expression operators. As in arithmetic expressions, there is an order of precedence among these operators:

Regular expression operator precedence

Precedence	Operator	Description
Highest	(), (?:), etc.	Parentheses and other grouping operators
	?, +, *, {*m*,*n*}, +?, etc.	Repetition
	^abc	Sequence (see below)
Lowest	|	Alternation

Fortunately, there are only four precedence levels—imagine if there were as many as there are for arithmetic expressions! Parentheses and the other grouping operators[1] have the highest precedence.

A *repetition* operator binds tightly to its argument, which is either a single atom or a grouping operator:

`ab*c` *Matches* ac, abc, abbc, abbbc, *etc.*

`abc*` *Matches* ab, abc, abcc, abccc, *etc.*

[1] A multitude of new grouping operators were introduced in Perl 5.

`ab(c)*`	*Same thing, and memorizes the* c *actually matched.*
`ab(?:c)*`	*Same thing, but doesn't memorize the* c.
`abc{2,4}`	*Matches* abcc, abccc, abcccc.
`(abc)*`	*Matches empty string*, abc, abcabc, *etc.; memorizes* abc.

Placing two atoms side by side is called *sequence*. Sequence is a kind of operator, even though it is written without punctuation. This is similar to the invisible multiplication operator in a mathematical expression like $x = ay + b$. To illustrate this, let's suppose that sequence were actually represented with the character "•". Then the above examples would look like:

`a•b*•c`	*Matches* ac, abc, abbc, abbbc, *etc.*
`a•b•c*`	*Matches* ab, abc, abcc, abccc, *etc.*
`a•b•(c)*`	*Same thing, and memorizes the* c *actually matched.*
`a•b•(?:c)*`	*Same thing, but doesn't memorize the* c.
`a•b•c{2,4}`	*Matches* abcc, abccc, abcccc.
`(a•b•c)*`	*Matches empty string*, abc, abcabc, *etc.; memorizes* abc.

The last entry in the precedence chart is *alternation*. Let's continue to use the "•" notation for a moment:

`e•d|j•o`	*Matches* ed *or* jo.
`(e•d)|(j•o)`	*Same thing.*
`e•(d|j)•o`	*Matches* edo *or* ejo.
`e•d|j•o{1,3}`	*Matches* ed, jo, joo, jooo.

The zero-width atoms, for example, ^ and \b, group in the same way as other atoms:

`^e•d|j•o$`	*Matches* ed *at beginning*, jo *at end.*
`^(e•d|j•o)$`	*Matches exactly* ed *or* jo.

It's easy to forget about precedence. Removing excess parentheses is a noble pursuit, especially within regular expressions, but be careful not to remove too many:

```
/^Sender|From:\s+(.*)/;            WRONG—would match:
                                    X-Not-Really-From: faker
                                    Senderella is misspelled
```

The pattern was meant to match `Sender:` and `From:` lines in a mail header, but it actually matches something somewhat different. Here it is with some parentheses added to clarify the precedence:

```
/(^Sender)|(From:\s+(.*))/;
```

Adding a pair of parentheses, or perhaps memory-free parentheses `(?:…)`, fixes the problem:

```
/^(Sender|From):\s+(.*)/;          $1 contains Sender or From.
                                   $2 has the data.

/^(?:Sender|From):\s+(.*)/;        $1 contains the data.
```

Double-quote interpolation

Perl regular expressions are subject to the same kind of interpolation that double-quoted strings are.[2] Interpolated variables and string escapes like `\U` and `\Q` are *not* regular expression atoms and are never seen by the regular expression parser. Interpolation takes place in a single pass that occurs before a regular expression is parsed:

```
/te(st)/;         Matches test in $_.
/\Ute(st)/;       Matches TEST.
/\Qte(st)/;       Matches te(st).

$x = 'test';
/$x*/;            Matches tes, test, testt, etc.
/test*/;          Same thing as /$x*/.
```

Double-quote interpolation and the separate regular expression parsing phase combine to produce a number of common “gotchas.” For example, here’s what can happen if you forget that an interpolated variable is not an atom:

Read a pattern into `$pat` *and match two consecutive occurrences of it.*

```
chomp($pat = <STDIN>);                  For example, bob.

print "matched\n" if /$pat{2}/;         WRONG—/bob{2}/.
```

[2] Well, more or less. The $ anchor receives special treatment so that it is not always interpreted as a scalar variable prefix.

```
print "matched\n" if /($pat){2}/;          RIGHT—/(bob){2}/.
print "matched\n" if /$pat$pat/;           Brute force way.
```

In this example, if the user types in bob, the first regular expression will match bobb, because the contents of $pat are expanded before the regular expression is interpreted.

All three regular expressions in this example have another potential pitfall. Suppose the user types in the string “hello :-)”. This will generate a fatal run-time error. The result of interpolating this string into /($pat){2}/ is /(hello :-)){2}/, which, aside from being nonsense, has unbalanced parentheses.

If you don’t want special characters like parentheses, asterisks, periods, and so forth interpreted as regular expression metacharacters, use the quotemeta operator or the quotemeta escape, \Q. Both quotemeta and \Q put a backslash in front of any character that isn’t a letter, digit, or underscore:

```
chomp($pat = <STDIN>);                      For example, hello :-).
$quoted = quotemeta $pat;                   Now hello\ \:\-\).

print "matched\n" if /($quoted){2}/;        “Safe” to match now.
print "matched\n" if /(\Q$pat\E){2}/;       Another approach.
```

As with seemingly everything else pertaining to regular expressions, tiny errors in quoting metacharacters can result in strange bugs:

```
print "matched\n" if /(\Q$pat){2}/;         WRONG—no \E ... means
                                            /hello \ \:\-\)\{2\}/.
```

Item 16: Use regular expression memory.

Although regular expressions are handy for determining whether a string looks like one thing or another, their greatest utility is in helping parse the contents of strings once a match has been found. To break apart strings with regular expressions, you must use regular expression memory.

The memory variables: $1, $2, $3, and so on

Most often, parsing with regular expressions involves the use of the regular expression *memory variables* $1, $2, $3, and so on. Memory variables are associated with parentheses inside regular expressions. Each pair of parentheses in a regular expression “memorizes” what its contents matched. For example:

```
$_ = 'http://www.perl.org/index.html';   Memorize hostname and
m#^http://([^/]+)(.*)#;                  path following http://.

print "host = $1\n";                     host = www.perl.org
print "path = $2\n";                     path = /index.html
```

Only *successful* matches affect the memory variables. An unsuccessful match leaves the memory variables alone, even if it appears that part of a match might be succeeding:

```
Continued from above:

$_ = 'ftp://ftp.uu.net/pub/';            ftp doesn't match http.
m#^http://([^/]+)(.*)#;                  Same pattern as above.

print "host = $1\n";                     Still www.perl.org.
print "path = $2\n";                     Still /index.html.
```

When a pair of parentheses matches several different places in a string, the corresponding memory variable contains the *last* match:

```
$_ = 'ftp://ftp.uu.net/pub/systems';     Last fragment of the path
m#^ftp://([^/]+)(/[^/]*)+#;              goes into $2.

print "host = $1\n";                     host = ftp.uu.net
print "fragment = $2\n";                 fragment = /systems
                                         but matched /pub first.
```

In cases involving nested parentheses, *count left parentheses* to determine which memory variable a particular set of parentheses refers to:

```
$_ = 'ftp://ftp.uu.net/pub/systems';     This pattern is similar to
m#^ftp://([^/]+)((/[^/]*)+)#;            the last one, but also
                                         collects the whole path.

print "host = $1\n";                     host = ftp.uu.net
print "path = $2\n";                     path = /pub/systems
print "fragment = $3\n";                 fragment = /systems
```

The “count left parentheses” rule applies to all regular expressions, even ones involving alternation:

```
$_ = 'ftp://ftp.uu.net/pub';             Just grab the first
m#^((http)|(ftp)|(file)):#;              (protocol) portion of a
                                         URL.

print "protocol = $1\n";                 protocol = ftp
print "http = $2\n";                     http =
print "ftp = $3\n";                      ftp = ftp
print "file = $4\n";                     file =
```

The $+ special variable contains the value of the last non-empty memory:

Continued from above:

```
print "\$+ = $+\n";                  $+ = ftp
```

The parade of frills continues! Memory variables are automatically localized by each new scope. In a unique twist, the localized variables receive *copies* of the values from the outer scope—this is in contrast to the usual reinitializing of a localized variable:

```
$_ = 'ftp://ftp.uu.net/pub';         Take a URL apart in two steps—
m#^([^:]+)://(.*)#;                  first, split off the protocol.

print "\$1, \$2 = $1, $2\n";         $1, $2 = ftp, ftp.uu.net/pub

{                                    Now, split into host and path.
  print "\$1, \$2 = $1, $2\n";       $1, $2 = ftp, ftp.uu.net/pub
  $2 =~ m#([^/]+)(.*)#;
  print "\$1, \$2 = $1, $2\n";       $1, $2 = ftp.uu.net, /pub
}

print "\$1, \$2 = $1, $2\n";         $1, $2 = ftp, ftp.uu.net/pub
                                     The old $1 and $2 are back.
```

The localizing mechanism used is `local`, not `my` (see Item 23).

Backreferences

Regular expressions can make use of the contents of memories via *backreferences*. The atoms \1, \2, \3, and so on match the *contents* of the corresponding memories. An obvious (but not necessarily useful) application of backreferences is solving simple word puzzles:

```
/(\w)\1/;                            Matches doubled word
                                     char—aa, 11, __.

/(\w)\1+/;                           2 or more—aaa, bb,
                                     222222.

/((\w)\2){2,}/;                      Consecutive pairs—aabb,
                                     22__66 ... remember
                                     "count left parentheses"
                                     rule.

/([aeiou]).*\1.*\1.*\1/;             Same vowel 4 times.
/([aeiou])(.*\1){3}/;                Another way.
/([aeiou]).*?\1.*?\1.*?\1/;          More efficient (see Item
                                     17).
```

This kind of thing is always good for 10 minutes of fun on a really slow day. Just sit at your Unix box and type things like:

```
% perl -ne 'print if /([aeiou])(.*\1){3}/' /usr/dict/words
```

I get 106 words from this one, including “tarantara.” Hmm.

Backreferences are a powerful feature, but you may not find yourself using them all that often. Sometimes they are handy for dealing with delimiters in a simplistic way:

```
/(['"]).*\1/;                          'stuff' or "stuff",
                                       greedy.

/(['"]).*?\1/;                         Non-greedy version (see
                                       Item 17).

/(['"])(\\\1|.)*?\1/;                  Handles escapes: \', \".
```

Unfortunately, this approach breaks down quickly—you can’t use it to match parentheses (even without worrying about nesting), and there are faster ways to deal with embedded escapes.

The match variables: $`, $&, $'

In addition to memory variables like $1, $2, and $3, there are three special *match variables* that refer to the match and the string from which it came. $& refers to the portion of the string that the entire pattern matched, $` refers to the portion of the string preceding the match, and $' refers to the portion of the string following the match. As with memory variables, they are set after each successful pattern match.

Match variables can help with certain types of substitutions in which a replacement string has to be computed:

Go through the contents of OLD *a line at a time, replacing some one-line HTML comments.*

```
while (<OLD>) {
  while (/<!--\s*(.*?)\s*-->/g) {       Extract info from
    $_ = $` . new_html($1) . $'         comment and check it out.
      if ok_to_replace($1);             Replace comment.
  }
  print NEW $_;
}
```

Some people complain that using match variables makes Perl programs run slower. This is true. There is some extra work involved in maintaining the values of the match variables, and once any of the match variables appears in a program, Perl maintains them for *every* regular expression match in the program. If you are concerned about speed, you may want to rewrite code that uses match variables. You can generally rephrase such code as substitutions that use memory variables. In the case above, you could do the obvious (but incorrect):

```
while (<OLD>) {
  while (/<!--\s*(.*?)\s*-->/) {
    s/<!--\s*(.*)\s*-->/new_html($1)/e
      if ok_to_replace($1);
  }
  print NEW $_;
}
```

Use substitution rather than match variables for replacement. However, /g *won't work; thus this is broken for lines that contain more than one comment.*

Or, a correct but slightly more involved alternative:

```
while (<OLD>) {
  s{(<!--\s*(.*?)\s*-->)}{
    ok_to_replace($2) ?
      new_html($2) : $1;
  }eg;
  print NEW $_;
}
```

Use s///eg *for replacement (looks better using braces as delimiters).*

In most cases, though, I would recommend that you write whatever makes your code clearer, including using match variables when appropriate. Worry about speed after everything works and you've made your deadline (see Item 22).

The localizing behavior of match variables is the same as that of memory variables.

Memory in substitutions

Memory and match variables are often used in substitutions. Uses of $1, $2, $&, and so on within the replacement string of a substitution refer to the memories from the match part, not an earlier statement (hopefully, this is obvious):

```
s/(\S+)\s+(\S+)/$2 $1/;
```

Swap two words.

```
%ent = (
  '&' => 'amp', '<' => 'lt',
  '>' => 'gt'
);
$html =~ s/([&<>])/&$ent{$1};/g;
```

Here is an approach to HTML entity escaping.

a&b *becomes* a&amp;b

```
$newsgroup =~ s/(\w)\w*/$1/g;
```

comp.sys.unix *becomes* c.s.u.

Some substitutions using memory variables can be accomplished without them if you look at what to throw away, rather than what to keep.

```
s/^\s*(.*)/$1/;
```

Eliminate leading whitespace, hard way.

```
s/^\s+//;
```

Much better!

```
$_ = "FOO=bar BLETCH=baz";
s/(FOO=\S+)|\w+=\S+/$1/g;
```

Throw away assignments except FOO=.

```
s/(this|that)|(\w)/$1\U$2/g;
```

Uppercase all words except this *and* that.

You can use the /e (eval) option to help solve some tricky problems:

```
s/(\S+\.txt)\b/-e $1 ? $1 :
  "<$1 not found>"/ge;
```

Replace all the nonexistent foo.txt.

Substitutions using /e can sometimes be more legibly written using matching delimiters and possibly the /x option (see Item 21):

```
s{
  (\S+\.txt)\b   # ending in .txt?
}{
  -e $1 ? $1 : "<$1 not found>"
}gex;
```

Same as above, written with /x *option to ignore whitespace (including comments) in pattern.*

Matching in a list context

In a list context, the match operator m// returns a list of values corresponding to the contents of the memory variables. If the match is unsuccessful, the match operator returns an empty list. This doesn't change the behavior of the match variables: $1, $2, $3, and so on are still set as usual.

Matching in a list context is one of the most useful features of the match operator. It allows you to scan and split apart a string in a single step:

```
($name, $value) = /^([^:\s]*):\s+(.*)/;
```

Parse an RFC822-like header line.

```
($bs, $subject) =
  /^subject:\s+(re:\s*)?(.*)/i;
```

Get the subject, minus leading re:.

```
$subject =
  (/^subject:\s+(re:\s*)?(.*)/i)[1];
```

Or, instead of a list assignment, a literal slice.

```
($mode, $fn) = /begin\s+(\d+)\s+(\S+)/i
```

Parse a uuencoded file's begin *line.*

Using a match inside a map is even more succinct. This is one of my favorite ultra-high-level constructs:

```
($date) =
  map { /^Date:\s+(.*)/ } @msg_hdr;
```

Find the date of a message in not very much Perl.

```
@protos =
  map { /^(\w+)\s+stream\s+tcp/ } <>;
print "protocols: @protos\n";
```

Produce a list of the named tcp stream protocols by parsing `inetd.conf` *or something similar.*

Note that it turns out to be extremely handy that a failed match returns an empty list.

A match with the /g option in a list context returns *all* the memories for each successful match:

```
print "fred quit door" =~ m/(..)\b/g;
```

Prints `editor` *— last two characters of each word.*

Memory-free parentheses

Parentheses in Perl regular expressions serve two different purposes: grouping and memory. Although this is usually convenient, or at least irrelevant, it can get in the way at times. Here's an example we just saw:

```
($bs, $subject) =
  /^subject:\s+(re:\s*)?(.*)/i;
```

Get the subject, minus leading `re:`*.*

We need the first set of parentheses for grouping (so the ? will work right), but they get in the way memory-wise. What we would like to have is the ability to group without memory. Perl 5 introduced a feature for this specific purpose. *Memory-free parentheses* (?:…) group like parentheses, but they don't create backreferences or memory variables:

```
($subject) =
  /^subject:\s+(?:re:\s*)?(.*)/i;
```

Get the subject, no bs.

Memory-free parentheses are also handy in the match-inside-map construct (see above), and for avoiding delimiter retention mode in `split` (see Item 19). In some cases they also may be noticeably faster than ordinary parentheses (see Item 22). On the other hand, memory-free parentheses are a pretty severe impediment to readability and probably are best avoided unless needed.

Tokenizing with regular expressions

Tokenizing or "lexing" a string—dividing it up into lexical elements like whitespace, numbers, identifiers, operators, and so on—offers an interesting application for regular expression memory.

If you have written or tried to write computer language parsers in Perl, you may have discovered that the task can seem downright hard at times. Perl seems to be missing some features that would make things easier. The problem is that when you are tokenizing a string, what you want is to find out *which* of several possible patterns matches the *beginning* of a string (or at a particular point in its middle). On the other hand, what Perl is good at is finding out *where* in a string a *single* pattern matches. The two don't map onto one another very well.

Let's take the example of parsing simple arithmetic expressions containing numbers, parentheses, and the operators +, -, *, and /. (Let's ignore whitespace, which we could have substituted or tr-ed out beforehand.) One way to do this might be:

```
while ($_) {
  if (/^(\d+)/) {
    push @tok, 'num', $1;
  } elsif (/^([+\-\/*()])/) {
    push @tok, 'punct', $1;
  } elsif (/^([\d\D])/) {
    die "invalid char $1 in input";
  }
  $_ = substr($_, length $1);
}
```

Tokenize contents of $_ *into array* @tok.

Chop off what we recognized and go back for more.

This turns out to be moderately efficient, even if it looks ugly. However, a tokenizer like this one will slow down considerably when fed long strings because of the substr operation at the end. You might think of keeping track of the current starting position in a variable named $pos and then doing something like:

```
if (substr($_, $pos) =~ /^(\d+)/) {
```

However, this do-it-yourself technique probably won't be much faster and may be slower on short strings.

One approach that works reasonably well, and that is not affected unduly by the length of the text to be lexed, relies on the behavior of the match operator's /g option in a scalar context—we'll call this a "scalar m//g match." Each time a scalar m//g match is executed, the regular expression engine starts looking for a match at the current "match position," generally after the end of the preceding match—analogous to the $pos variable mentioned above. In fact, the current match position can be accessed (and changed) through Perl's pos operator. Applying a scalar m//g match allows you to use a single regular expression, and it frees you from having to keep track of the current position explicitly:

```
while (/                                           Use a match with the /g
  (\d+) |  # number                                option. The /x option is
  ([+\-\/*()]) |  # punctuation                    also used to improve
  ([\d\D])  # something else                       readability (see Item 21).
/xg) {
  if ($1 ne "") {                                  Examine $1, $2, $3 to see
    push @tok, 'num', $1;                          what was matched.
  } elsif ($2 ne "") {
    push @tok, 'punct', $2;
  } else {
    die "invalid char $3 in input";
  }
}
```

The most recent versions of Perl support a /c option for matches, which modifies the way scalar m//g operates. Normally, when a scalar m//g match *fails*, the match position is reset, and the next scalar m//g will start matching at the beginning of the target string. The /c option causes the match position to be *retained* following an unsuccessful match. This, combined with the \G anchor, which forces a match beginning at the last match position, allows you to write more straightforward tokenizers:

```
{                                                  A naked block for looping.
  if (/\G(\d+)/gc) {                               Is it a number?
    push @tok, 'num', $1;
  } elsif (/\G([+\-\/*()])/gc) {                   Is it punctuation?
    push @tok, 'punct', $1;
  } elsif (/\G([\d\D])/gc) {                       It's something else.
    die "invalid char $1 in input";
  } else {                                         Out of string?
    last;                                          We're done.
  }
  redo;                                            Otherwise, loop.
}
```

Although it isn't possible to write a single regular expression that matches nested delimiters, with scalar m//gc you can come fairly close:.

■ **Find nested delimiters using scalar m//gc.**

Here is an approach to matching nested braces. {qw({ 1 } -1)} *is an anonymous hash ref—it could have been written less succinctly as* {('{' => 1, '}' => -1)}.

```
$_ = " Here are { nested {} { braces } }!";        Input goes into $_.
```

■ **Find nested delimiters using scalar `m//gc`. (cont'd)**

```
{                                                        $c counts braces.
  my $c;
  while (/([{}])/gc) {                                   Find braces
    last unless ($c += {qw({ 1 } -1)}->{$1}) > 0         and count them
  };                                                     until count is 0.
}
print substr substr($_, 0, pos()), index($_, "{");       Print found string.
```

Item 17: Avoid greed when parsimony is best.

One of the stickier problems you may encounter in dealing with regular expressions is *greed*.

Greed isn't about money, at least where regular expressions are concerned. It's the term used to describe the matching behavior of most regular expression engines, Perl's included. A general rule[3] is that a Perl regular expression will return the *longest* match it can find, at the *first* position in a string at which it can find a match of any length. Repetition operators like `*` and `+` "gobble up" characters in the string until matching more characters causes the match to fail:

```
$_ = "Greetings, planet Earth!\n";          Some data to match.

/\w+/;                                      Matches Greetings.
/\w*/;                                      Matches Greetings.

/n[et]*/;                                   Matches n in Greetings.
/n[et]+/;                                   Matches net in planet.

/G.*t/;                                     Matches Greetings,
                                            planet Eart.
```

This is normally a desirable behavior. But not always. Be especially careful when using greedy regular expressions to match delimited patterns like quoted strings and C comments:

▼ **Don't use greedy regular expressions with delimiters.**

These examples illustrate incorrect patterns for matching text enclosed by delimiters—in this case single-quoted strings and C comments.

```
$_ = "This 'test' isn't successful?";       Hoping to match 'test'.

($str) = /('.*')/;                          Matches 'test' isn'.
```

[3] But not strictly accurate, as you will see.

▼ Don’t use greedy regular expressions with delimiters. (cont’d)

```
$_ = "/* temp */ x = 10; /* too much? */";    Hoping to match /* temp */.

s#(/\*.*\*/)##;                               OOPS—erases the whole string!
```

In these examples, Perl keeps matching beyond what appears to be the end of the pattern. But the match operator hasn’t run amok: ', /, and * are all matched by ., and the match ends at the last occurrence of ', or */. We can fix the single-quoted string example by excluding single quotes from the characters allowed inside the string:

```
$_ = "This 'test' isn't successful?";
($str) = /('[^']*')/;                         Matches 'test' now.
```

Straightening out the regular expression for C comments is more troublesome. I will bet confidently that when you write your first regular expression that you *believe* matches C comments, it will not work. Here is one of many possibilities—it seems reasonable at first:

```
s#/\*([^*]|\*[^/])*\*/##g;                    ALMOST works.
```

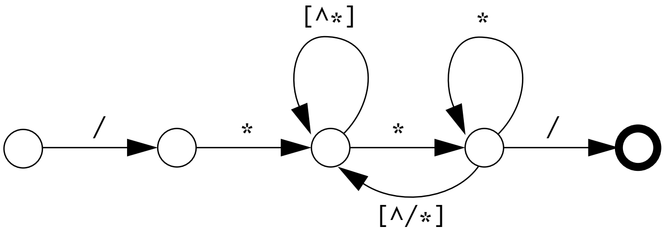
Do you see the problem with it? It fails on the following input:

```
/***/
```

The reason is that there is no way for it to match an asterisk inside the comment that isn’t followed by exactly one other character, thus an odd number of asterisks fails to match. It has other problems, too, but this one is enough. The real answer looks like this:[4]

```
s#/\*[^*]*\*+([^/*][^*]*\*+)*/##g;            CORRECT
```

You are not likely to understand the how and why of this without recourse to a diagram of the underlying state machine:



[4] As long as it isn’t necessary to deal with things like comments embedded in strings, anyway.

And if you haven't suffered through a class in discrete math or compiler construction, this may not help you either. In fact, a good many people will go through their lives using regular expressions like the above without knowing why they work. This isn't necessarily bad; it's just not ideal.

Now, if it's this hard to construct a regular expression for something as simple as C comments, imagine what it could be like to try to write one for something more complex, like HTML comments, or strings with character escapes. Pretty scary.

Fortunately, Perl 5 has *non-greedy* repetition operators. This is a powerful and enormously helpful new feature that allows you to write simple regular expressions for cases that previously required complex or even impossibly difficult regular expressions.

You can make any repetition operator (*, +, {*m*,*n*}) non-greedy by following it with a question mark. The operator will now match the *shortest* string that results in a pattern match, rather than the longest. This makes the examples above trivially simple:

■ **Do use non-greedy regular expressions with delimiters.**

These examples illustrate patterns that correctly match text enclosed by delimiters.

```
$_ = "This 'test' isn't successful?";
($str) = /('.*?')/;                          Matches 'test'.

$_ = "/* temp */ x = 10; /* too much? */";
s#(/\*.*?\*/)##;                             Deletes /* temp */.
```

You can now attempt more ambitious things, like a double-quoted string with character escapes (let's support \", \\, and \123):

```
$_ = 'a "double-q \"string\042"';
($str) = /("(\\["\\]|\\\d{1,3}|.)*?")/;
print $str;                                "double-q \"string\042"
```

The only problem with non-greedy matching is that it can be slower than greedy matching. Don't use non-greedy operators unnecessarily. But *do* use non-greedy operators to avoid having to write complex regular expressions that might or might not be correct.

Procedural regular expressions versus deterministic finite automatons (DFAs)

Perl and most other tools with robust pattern-matching capabilities that include features like backreferences use what I call a *procedural* approach to regular expression pattern matching. When Perl encounters a regular expression in a program, it compiles it into a treelike structure and saves it away. When that regular expression is used to match a string, Perl looks for the match by "executing" the compiled regular expression. Consider a simple regular expression and target string:

```
$_ = 'testing';
/t(e|es)./;
print "matched: $&\n";                          matched: tes
```

If Perl could talk, it might describe the matching process something like this:

"OK, start at first character position. Looking for a t. Got one.

"Now, an alternation, first one is e. Looking for e. Got one.

"OK, the alternation matched. Next thing is a dot. Need one char to match the dot. Got an s.

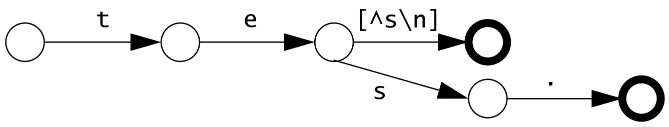
"Anything else? Nope. Guess we're done."

If you have no background experience with tools like *lex* or *flex*, or if this is the only kind of regular expression you have ever known, you probably don't see anything unusual with this interpretation of this regular expression. On the other hand, if you are familiar with *flex*, you might be thinking, "Hmm, why didn't that match test instead?"

Well, you *could* get it to match test by rewriting it:

```
$_ = 'testing';
/t(es|e)./;
print "matched: $&\n";                          matched: test
```

This illustrates the difference—outwardly, at least—between procedural regular expressions and state machine or DFA regular expressions.[5] Tools like *flex* go far beyond parsing regular expressions. They go through an involved process that generates a deterministic state machine from the regular expression, rendering it into what is basically a table of numbers. If you use *flex* on the regular expression above, it will generate an internal state machine that looks something like this:



[5] The Hip Owls book uses the term "NFA regular expressions" to refer to what I call "procedural" matching.

The bold circles represent "accepting states" in which a complete match has been found. This view of the regular expression is somewhat different from the one that Perl has. For one thing, this DFA would have matched `test`, not `tes`, from the string `testing`. For another, the process of creating a DFA from a regular expression discards the syntactic structure of the original regular expression. In general, DFA-based tools always find the longest match for a regular expression, regardless of the order of alternations or grouping of repetition operators. On the other hand, as you can see, the arrangement of the parts of a Perl regular expression *is* significant. The flexibility to change what a pattern matches by rearranging its parts helps make Perl regular expressions particularly powerful and expressive.

If you are more familiar with the DFA view of regular expressions than the procedural view, you should take some time to think about procedural regular expressions. Experiment with them. You can do things with procedural regular expressions that are very difficult or impossible to do with DFA regular expressions, especially if you make use of Perl's non-greedy repetition operators.

The down side to procedural regular expressions is that they generally run slower than DFA regular expressions. But they are not *that* much slower, and they do compile into an internal represention significantly faster.

Item 18: Remember that whitespace is not a word boundary.

You will frequently use the set of whitespace characters, `\s`, the set of word characters, `\w`, and the word boundary anchor, `\b`, in your Perl regular expressions. Yet you should be careful when you use them *together*. Consider the following pattern match:

```
@who = `who`;                        donna  pts/3  Oct  1 18:33
$_ = pop @who;
($user, $tty) = /(\w+)\s+(\w+)/;     It looks innocuous at first.
```

This works fine on input like "`joebloe ttyp0`…". However, it will not match at all on strings like "`webmaster-1 ttyp1`…" and will return a strange result on "`joebloe pts/10`…". This match probably should have been written:

```
($user, $tty) = /(\S+)\s+(\S+)/;       BETTER—\S next to \s.
```

There is probably something wrong in your regular expression if you have `\w` adjacent to `\s`, or `\W` adjacent to `\S`. At the least, you should examine such regular expressions *very* carefully.

from *Effective Perl Programming*, by Joseph N. Hall with Randal L. Schwartz

Another thing to watch out for is a “word” that contains punctuation characters. Suppose you want to search for a whole word in a text string:

```
print "word to search for: ";
$word = <STDIN>;
print "found\n" if
    $text =~ /\b\Q$word\E\b/;
```

Hmm—are word boundaries what you want?

This works fine for input like `hacker` and even `Perl5-Porter`, but fails for words like `goin'`, or any word that does not begin and end with a `\w` character. It also will consider `isn` a matchable word if `$text` contains `isn't`. The reason is that `\b` matches transitions between `\w` and `\W` characters—not transitions between `\s` and `\S` characters. If you want to support searching for words delimited by whitespace, you will have to write something like this instead:

```
print "word to search for: ";
$word = <STDIN>;
print "found\n" if
    $text =~ /(^|\s)\Q$word\E($|\s)/;
```

BETTER—use whitespace as a delimiter.

The word boundary anchor, `\b`, and its inverse, `\B`, are zero-width patterns. Even though they are not the only zero-width patterns (`^`, `\A`, etc. are others), they are the hardest to understand. If you are not sure what `\b` and `\B` will match in your string, try substituting for them:

```
$text = "What's a \"word\" boundary?";
($btext = $text) =~ s/\b/:/g;
($Btext = $text) =~ s/\B/:/g;
print "$btext\n$Btext\n";
```

Insert colon at word boundaries and not-word boundaries.

```
% tryme
:What:':s: :a: ":word:" :boundary:?
W:h:a:t's a :"w:o:r:d": b:o:u:n:d:a:r:y?:
```

The results at the ends of the string should be especially interesting to you. Note that if the last (or first) character in a string is not a `\w` character, there is no word boundary at the end of the string. Note also that there are not-word boundaries between consecutive `\W` characters (like space and double quote) as well as consecutive `\w` characters.

Matching at the end of line: $, \Z, /s, /m

Of course, `$` matches at the end of a line—or does it? Officially, it matches at the end of the string being matched, *or* just before a final newline occurring at the end of the string. This feature makes it easy to match newline-terminated data:

```
print "some text\n" =~ /(.*)$/;          Prints "some text", as if
                                         newline wasn't there.

print "some text" =~ /(.*)$/;            Same thing.
```

The /s (single-line—sort of) option changes the meaning of . (period) so that it matches any character instead of any character but newline. This is useful if you want to capture newlines inside a string:

```
print "2\nlines\n" =~ /(.*)/;            2 (Period won't match
                                         newline.)

print "2\nlines\n" =~ /(.*)/s;           2\nlines\n
```

However, /s *does not* change the meaning of $:

```
$_ = "some text\n";                      Yields some tex<end>\n.
s/.$/<end>/s;                            (Replaces the character
                                         before \n.)
```

To force $ to really match the end of the string, you need to be more insistent. One way to do this is to use the (?!…) regular expression operator:

```
$_ = "some text\n";                      Now yields
s/.$(?!\n)/<end>/s;                      some text<end>.
```

Here, (?!\n) ensures that there are no newlines after the $.[6]

Ordinarily, $ only matches before the end of the string or a trailing newline. However, the /m (multi-line) option modifies the operation of $ so that it can also match before intermediate newlines. The /m option also modifies ^ so that it will match a position immediately following a newline in the middle of the string:

```
$_ = "2\nlines";
s/^/<start>/mg;                          <start>2\n<start>lines

$_ = "2\nlines";
s/$/<end>/mg;                            2<end>\nlines<end>
```

[6] In earlier versions of Perl you may have to surround the $ with memory-free parentheses—(?:$) instead of $—since the regular expression parser recognizes $(as a special variable. This behavior was recently changed so that $ preceding (is now recognized as an anchor, not part of a variable—as has long been the case with $ preceding).

```
%scores =
  <<'EOF' =~ /^(.*?):\s*(.*)/mg;
fred: 205
barney: 195
dino: 30
EOF
```

```
%scores = (
 'fred' => 205,
 'barney' => 195,
 'dino' => 30
);
```

(See Item 13 for more about here-doc strings.)

The `\A` and `\Z` anchors retain the original meanings of `^` and `$`, respectively, whether or not the `/m` option is used:

```
$_ = "2\nlines";
s/\A/<start>/mg;                    <start>2\nlines

$_ = "2\nlines";
s/\Z/<end>/mg;                      2\nlines<end>
```

Item 19: Use split for clarity, unpack for efficiency.

The elegance of list assignments in Perl is infectious, especially when combined with pattern matches. As you start using both features, you may find yourself writing code like:

```
($a, $b, $c) =
  /^(\S+)\s+(\S+)\s+(\S+)/;
```

Get first 3 fields of `$_`.

Of course, this is a natural application for `split`:

```
($a, $b, $c) = split /\s+/, $_;
```

Get first 3 fields of `$_`.

```
($a, $b, $c) = split;
```

Splits `$_` *on whitespace by default.*

The two approaches take about the same amount of time to run, but the code using `split` is simpler.

You can use pattern matches for more complex chores:

```
($a) =
  /[^:]*:[^:]*:[^:]*:[^:]*:([^:])/;
```

Get 5th field of `$_` *(delimited by colons).*

```
($a) = /(?:[^:]*:){4}([^:])/;
```

Another way to do it.

Using `split`, we have the alternative:

```
($a) = (split /:/)[4];
```

Get 5th field of `$_` *(delimited by colons).*

If you go to the trouble to benchmark these examples, you may find that the version using a pattern match runs significantly faster than the version using `split`. This wouldn't be a problem, except that the pattern match is significantly harder to read and understand. This is a general rule—pattern matches tend to be faster, and `split` tends to be simpler and easier to read. In cases like this, you have a decision to make. Do you use the faster code, or do you use the code that is easier to understand? I think the choice is obvious. If you must have speed, use a pattern match. But in general, readability comes first. If speed is not the most important issue, use `split` whenever the problem fits it.

List slices work effectively in combination with `split`:

```
$_ = "/my/file/path";                      Get whatever follows the
$basename = (split /\//, $_)[-1];          last /, or the whole thing.
```

You can use `split` several times to divide a string into successively smaller pieces. For example, suppose that you have a line from a Unix `passwd` file whose fifth field (the "GCOS" field) contains something like `"Joseph N. Hall, 555-2345, Room 888"`, and you would like to pick out just the last name:

```
($gcos) = (split /:/)[4];                  Fifth field in $gcos.
($name) = (split /,/, $gcos);              Stuff before , in $name.
($last) = (split / /, $name)[-1];          Last name in $last.
```

There are some situations where `split` can yield elegant solutions. Consider one of our favorite problems, matching and removing C comments from a string. You could use `split` to chop such a string up into a list of comment delimiters and whatever appears between them, then process the result to toss out the comments:

■ **Use `split` to process strings containing multi-character delimiters.**

The following code will print `$_` with C comments removed. It deals with double-quoted strings that possibly contain comment delimiters. The memory parentheses in the `split` pattern cause the delimiters, as well as the parts between them, to be returned.

```
for (split m!("(:?\\\W|.)*?"|/\*|\*/)!) {    Split on strings and delimiters.
  if ($in_comment) {
    $in_comment = 0 if $_ eq "*/"             Look for */ if in a comment.
  } else {
    if ($_ eq "/*") {                         Look for /* if not in a comment.
      $in_comment = 1;
      print " ";                              Comments become a space.
    } else {
      print;                                  If not in a comment, print.
    }
  }
}
```

from *Effective Perl Programming*, by Joseph N. Hall with Randal L. Schwartz

Handling columns with unpack

From time to time, you may encounter input that is organized in columns. Although you can use pattern matches to divide data up by columns, the unpack operator (see Item 53) provides a much more natural and efficient mechanism.

Let's take some output from a typical ps command line (a few columns have been lopped off the right so the output will fit here):

```
% ps l
 F   UID   PID  PPID CP PRI NI   SZ  RSS    WCHAN S TT
 8   100  7363  7352  0  48 20 1916 1492 write3ve S pts/3
 8   100 14227  7363  0  58 20  868  704 write3ve S pts/3
 8   998 28693  3327  0  58 20 3068 1724          T pts/2
```

Here, a split on whitespace would be ineffective, because the fields are determined on a column basis, not on a whitespace basis. Note that the WCHAN field doesn't even exist for the last line. This is a good time to trundle out the unpack operator.

■ **Use unpack to process column-delimited data.**

The following example extracts a few fields from the output of the ps command and prints them.

```
chomp (@ps = `ps l`);                         Collect some output.

shift @ps;                                    Toss first line.
for (@ps) {
  ($uid, $pid, $sz, $tt) =                    Unpack data and print it.
    unpack '@3 A5 @8 A6 @30 A5 @52 A5', $_;
  print "$uid $pid $sz $tt\n";
}
```

Note that the @ specifier does not return a value. It moves to an absolute position in the string being unpacked. In the example above, "@8 A6" means six characters starting at position 8.

You may find it aggravating to have to manually count out columns for the unpack format. The following program may help you get the right numbers with less effort:

Put a "picture" of the input in $_, and this program will generate a format.

```
$_ =
  '   aaaaabbbbbb                ccccc                 ddddd';
```

```
while (/(\w)\1+/g) {
    print '@' . length($`) . ' A' . length($&) . ' ';
}
print "\n";
```

You could also experiment interactively with the debugger (see Item 39) to find the correct column numbers.

Item 20: Avoid using regular expressions for simple string operations.

Regular expressions are wonderful, but they are not the most efficient way to perform all string operations. Although regular expressions can be used to perform string operations like extracting substrings and translating characters, they are better suited for more complex operations. Simple string operations in Perl should be handled by special-purpose operators like `index`, `rindex`, `substr`, and `tr///`.

Bear in mind that all regular expression matches, even simple ones, have to manipulate memory variables. If all you need is a comparison or a substring, manipulating memory variables is a waste of time. For this reason, if no other, you should prefer special-purpose string operators to regular expression matches whenever possible.

Compare strings with string comparison operators

If you have two strings to compare for equality, use string comparison operators, not regular expressions:

```
do_it() if $answer eq 'yes';
```

Fastest way to compare strings for equality.

The string comparison operators are at least twice as fast as regular expression matches:

```
do_it() if $answer =~ /^yes$/;
```

Slower.

```
do_it() if $answer =~ /yes/;
```

Even slower, and probably wrong, without anchors; e.g. on `"my eyes hurt"`.

A few more complex comparisons are also faster if you avoid regular expressions:

```
do_it() if lc($answer) eq 'yes';
```

Faster.

```
do_it() if $answer =~ /^yes$/i;
```

Slower.

Find substrings with index and rindex

The index operator locates an occurrence of a shorter string in a longer string. The rindex operator locates the rightmost occurrence, still counting character positions from the left:

```
$_ = "It's a Perl Perl Perl Perl World.";

$left = index $_, 'Perl';                7
$right = rindex $_, 'Perl';              22
```

The index operator is very fast—it uses a Boyer-Moore algorithm for its searches. Perl will also compile index-like regular expressions into Boyer-Moore searches. You could write:

Continued from above:

```
/Perl/;                                  Slow, with a gratuitous
$left = length $';                       use of $'.
```

or, avoiding the use of $' (see Item 16 and Item 21):

```
$perl = 'Perl';                          Yes, the pos operator does
/$perl/og;                               have uses. This is still
$left = pos($_) - length($perl);         slow, though.
```

However, the overhead associated with using a regular expression match makes index several times faster than m// for short strings.

Extract and modify substrings with substr

The substr operator extracts a portion of a string, given a starting position and (optional) length:

```
$str = "It's a Perl World.";

print substr($str, 7, 4), "\n";          Perl
print substr($str, 7), "\n";             Perl World
```

The substr operator is much faster than a regular expression written to do the same thing (also see Item 19):

Continued from above:

```
print ($str =~ /^.{7}(.{4})/), "\n";     Perl — but yuck!
```

The nifty thing about substr is that you can make replacements with it by using it on the *left side* of an expression. The text referred to by substr is replaced by the string value of the right-hand side:

Continued from above:

```
substr($str, 7, 4) = "Small";              It’s a Small World.
```

You can combine `index` and `substr` to perform `s///`-like substitutions, but in this case `s///` is usually faster:

```
$str = "It's a Perl World.";

substr($str, index($str, 'Perl'), 4) =     It’s a Mad Mad Mad Mad
  "Mad Mad Mad Mad";                       World.

$str =~ s/Perl/Mad Mad Mad Mad/;           Less noisy, and probably
                                           faster.
```

You can also do other lvalue-ish things with a `substr`, such as binding it to substitutions or `tr///`:

```
$str = "It's a Perl World.";

substr($str, index($str, 'Perl'), 4)       It’s a PERL World.
  =~ tr/a-z/A-Z/;
```

Transliterate characters with tr///

Although it is possible to perform character-level substitutions with regular expressions, the `tr///` operator provides a much more efficient mechanism:

■ **Use tr///, not regular expressions, to transliterate characters.**

```
$_ = "secret message";

tr/n-za-m/a-z/;                            frperg zrffntr—string “rot13”
                                           encoded.

@h{'N'..'Z','A'..'M'} = ('a'..'z');        Over 20 times slower, not
s/([a-z])/$h{$1}/g;                        counting initializing the hash!
```

The `tr///` operator has other uses as well. It is the fastest way to count characters in a string, and it can be used to remove duplicated characters:

```
$digits = tr/0-9//;                        Count digits in $_, fast.

tr/ \n\r\t\f/ /s;                          Repeated whitespace
                                           becomes single space.

$_ = "Totally\r\nDOS\r\n";                 Convert DOS text file to
tr/\r//d;                                  Unix.
```

Item 21: Make regular expressions readable.

Regular expressions are often messy and confusing. There's no denying it—it's true.

One reason that regular expressions are confusing is that they have a very compact and visually distracting appearance. They are a "little language" unto themselves. However, this little language isn't made up of words like `foreach` and `while`. Instead, it uses atoms and operators like `\w`, `[a-z]` and `+`.

Another reason is that what regular expressions do can be confusing in and of itself. Ordinary programming chores generally translate more or less directly into code. You might think "count from 1 to 10" and write `for $i (1..10) { print "$i\n" }`. But a regular expression that accomplishes a particular task may not look a whole lot like a series of straightforward instructions. You might think "find me a single-quoted string" and wind up with something like `/'(?:\\'|.)*?'/`.

It's a good idea to try to make regular expressions more readable, especially if you intend to share your programs with others, or if you plan to work on them some more yourself at a later date. Of course, trying to keep regular expressions *simple* is a start, but there are a couple of Perl features you can use that will help you make even complex regular expressions more understandable.

Use /x to add whitespace to regular expressions

Normally, whitespace encountered in a regular expression is significant:

```
($a, $b, $c) = /^(\w+) (\w+) (\w+)/;     Find three words
                                          separated by one space, at
                                          start of $_.

$_ = "Testing                             $_ contains embedded
one                                       newlines (same as if we
two";                                     had used \n).

s/                                        Replace newlines with
/<lf>/g;                                  <lf>

print "$_\n";                             Testing<lf>one<lf>two
```

The `/x` flag, which can be applied to both pattern matches and substitutions, causes the regular expression parser to ignore whitespace (so long as it isn't preceded by a backslash, or isn't inside a character class), including comments:

```
($str) = /( ' (?: \\' | . )*? ' )/x;      Find a single-quoted
                                          string, including escaped
                                          quotes.
```

This can be especially helpful when a regular expression includes a complex alternation:

```
($str) = / (                              Find a double-quoted
  " (?:                                   string, including hex and
    \\\W | # special char                 octal escapes.
    \\x[0-9a-fA-F][0-9a-fA-F] | # hex
    \\[0-3]?[0-7]?[0-7] | # octal
    [^"\\] # ordinary char
  )* "
) /x;
```

Break complex regular expressions into pieces

As you saw in Item 15, regular expressions are subject to double-quote interpolation. You can use this feature to write regular expressions that are built up with variables. In some cases, this may make them easier to read:

```
$num = '[0-9]+';                          Create some
$word = '[a-zA-Z_]+';                     "subpatterns."
$space = '[ ]+';

$_ = "Testing 1 2 3";                     Some sample data.
@split = /($num | $word | $space)/gxo;    Match into an array.
print join(":", @split), "\n";            Testing: :1: :2: :3
```

The pattern this example creates is `/([0-9]+ | [a-zA-Z_]+ | [ ])/gxo`. We used the `/o` ("compile once") flag, because there is no need for Perl to compile this regular expression more than once.

Notice that there weren't any backslashes in the example. It's hard to avoid using backslashes in more complex patterns. However, because of the way Perl handles backslashes and character escapes in strings (and regular expressions), backslashes must be doubled to work properly:

```
$num = '\\d+';                            '\\d+' becomes the string
$word = '\\w+';                           '\d+', etc.
$space = '\\ +';

$_ = "Testing 1 2 3";                     Some sample data.
@split = /($num | $word | $space)/gxo;    Match into an array.
print join(":", @split), "\n";            Testing: :1: :2: :3
```

The pattern this example creates is `/(\d+ | \w+ | \ +)/gxo`.

If we want a literal backslash in a regular expression, it has to be backslashed (e.g., `/\\/` matches a single backslash). Because backslashes in variables have to be doubled, this can result in some ugly looking strings—`'\\\\'` to match a backslash and `'\\\\\\w'` to match a backslash followed by a `\w` character. This is not going to make our regular expressions more readable in any obvious way, so when dealing with subpatterns containing backslashes, it's wise to make up some strings in variables to hide this ugliness. Let's rewrite the double-quoted string example from above, this time using some variables:[7]

```
$back = '\\\\';                             Pattern for backslash.

$spec_ch = "$back\\W";                      Escaped char like \", \$.
$hex_ch = "${back}x[0-9a-fA-F]{2}";         Hex escape: \xab.
$oct_ch = "${back}[0-3]?[0-7]?[0-7]";       Oct escape: \123.
$char = "[^\"$back]";                       Ordinary char.

($str) = /(                                 Here's the actual pattern
 " (                                        match.
  $spec_ch | $hex_ch | $oct_ch | $char
 )* "
)/xo;
```

If you are curious as to exactly what a regular expression built up in this manner looks like, print it out. Here's one way:

Continued from above:

```
print <<EOT;                                Just wrap everything in a
/(                                          double-quoted here-doc
 " (                                        string.
  $spec_ch | $hex_ch | $oct_ch | $char
 )* "
)/xo;
EOT
```

This will print:

```
/(
 " (
  \\\W | \\x[0-9a-fA-F]{2} | \\[0-3]?[0-7]?[0-7] | [^"\\]
 )* "
)/xo;
```

[7] If something like `"${back}[0-3][0-7]{2}"` worries you, feel free to write it as `$back . "[0-3][0-7]{2}"`.

This is a fairly straightforward example of using variables to construct regular expressions. See the Hip Owls book for a much more complex example—a regular expression that can parse an RFC822 address.

Item 22: Make regular expressions efficient.

Although Perl's regular expression engine contains many optimizations for efficiency, it's possible—and easy at times—to write matches and substitutions that run much slower than they should.

Efficiency may not always be your primary objective. In fact, efficiency should *rarely* be a primary objective in software development. Generally, a programmer's first priority should be to develop adequate, robust solutions to problems. It doesn't hurt, though, to keep efficiency in mind.

Let's look at a few common issues for regular expressions in Perl. The list below is by no means exhaustive, but it's a start, and it should get you headed in the right direction.

Compile once with /o

The regular expressions for most pattern matches and substitutions are compiled into Perl's internal form only once—at compile time, along with the rest of the surrounding statements:

The pattern `/\bmagic_word\b/` *is compiled only once, since it remains constant. The compiled form is then used over and over again at run time.*

```
foreach (@big_long_list) {
  $count += /\bmagic_word\b/;
}
```

Count occurrences of `magic_word` *in* `@big_long_list`.

When a pattern contains interpolated variables, however, Perl recompiles it *every time it is used*:

The pattern `/\b$magic\b/` *is recompiled every time it is used in a match, since it contains an interpolated variable.*

```
print "give me the magic word: ";
chomp($magic = <STDIN>);
foreach (@big_long_list) {
  $count += /\b$magic\b/;
}
```

Count occurrences of the magic word in `@big_long_list`.

The reason for this behavior is that the variables making up the pattern might have changed since the last time the pattern was compiled, and thus the pattern itself might be different. Perl makes this assumption to be safe, but such recompilation is often unnecessary. In many cases, like the

/\b$magic\b/ example above, variables are used to construct a pattern that will remain the same throughout the execution of the program containing it. To recompile such a pattern each time it is used in a match is grossly wasteful. This problem arises often, and naturally there is a feature in Perl to help you solve it. Perl's /o ("compile once") flag causes a regular expression containing variables to be compiled *only once*—the first time it is encountered at run time:

■ **Use /o to compile patterns only once.**

The pattern /\b$magic\b/o *is compiled on the first iteration of the* foreach *loop, using whatever the value of* $magic *is at that time. The pattern is never compiled again, even if the value of* $magic *changes.*

```
print "give me the magic word: ";
chomp($magic = <STDIN>);
foreach (@big_long_list) {
  $count += /\b$magic\b/o;
}
```

Count occurrences of the magic word in @big_long_list*—note added* /o*.*

The /o flag also works for substitutions. Note that the replacement string in the substitution continues to work as it normally does—it can vary from match to match:

```
print "give me the magic word: ";
chomp($magic = <STDIN>);
foreach (@big_long_list) {
  s/\b$magic\b/rand_word()/eo;
}
```

Replace occurrences of $magic *with something returned by* rand_word()*. See also examples at end of Item 29.*

Don't use match variables

I mentioned in Item 16 that the match variables ($`, $&, and $') impose a speed penalty on your Perl programs. Whenever a Perl program uses one of the match variables, Perl has to keep track of the values of the match variables for *every single pattern match in the program*.

▼ **Don't use match variables ($`, $&, $') if speed is important.**

Using a match variable anywhere in your program activates a feature that makes copies of the match ($&), before-match ($`) and after-match ($') strings for every single match in the program.

```
$_ = "match variable";
/.*/;
print "Gratuitious use of a $&\n";
```

Uh-oh: We activated the match variable feature.

▼ **Don’t use match variables ($`, $&, $') if speed is important. (cont’d)**

```
while (<>) {
  push @merlyn, $_ if /\bmerlyn\b/;
}
```

This now runs slower because of the use of $& *above!*

Perl isn’t smart enough to know which pattern(s) the match variables might be referring to, so Perl sets up the values of the match variables every time it does a pattern match. This results in a lot of extra copying and unnecessary shuffling around of bytes.

Fortunately, the penalty isn’t that severe. In most cases (particularly if some I/O is involved, as above), your program will run only slightly slower, if at all. In test cases designed to spot the penalty, the extra time consumed can range from a few percent to 30 to 40 percent. Jeffrey Friedl reports a contrived test case in which the run time with a match variable present was 700 times longer, but it is unlikely you will face a situation like this.

Avoid unnecessary alternation

Alternation in regular expressions is generally slow. Because of the way the regular expression engine in Perl works, each time an alternative in a regular expression fails to match, the engine has to “backtrack” (see the next subheading) in the string and try the next alternative:

The pattern match below finds a word boundary, then tries to match george. *If that fails, it backs up to the boundary and tries to match* jane. *If that fails, it tries* judy, *then* elroy. *If a match is found, it looks for another word boundary.*

```
while (<>) {
  print if
    /\b(george|jane|judy|elroy)\b/;
}
```

There are some instances in which alternation is completely unnecessary. In these cases, it is usually vastly slower than the correct alternative. The classic mistake is using alternation instead of a character class:

▼ **Don’t use alternation (a|b|c) instead of a character class ([abc]).**

Using an alternation instead of a character class can impose a tremendous speed penalty on a pattern match.

```
while (<>) {
  push @var, m'((?:\$|@|%|&)\w+)'g;
}
```

Look for Perl variable-name-like things. Single quote delimiters turn off variable interpolation inside pattern.

▼ Don’t use alternation (a|b|c) instead of a character class ([abc]). (cont’d)

```
while (<>) {
  push @var, m'([$@%&]\w+)'g;
}
```

Look for Perl variable-name-like things. This is about four times faster than the version using alternation.

Avoid unnecessary backtracking

Perl’s procedural regular expression engine (see Item 17) works by stepping through a compiled version of a pattern, in effect using it as if it were a little program trying to match pieces of text:

- When you write a sequence, you are creating instructions that mean “try to match this, followed by that, followed by . . .”
- When you write an alternation, you are creating instructions that mean “try to match this first; if that doesn’t work, back up and try to match that; if that doesn’t work . . .” and so on.
- When you use a repetition operator like + or *, you are instructing the engine to “try to find as many of these in a row as you can.”

Consider the pattern `/\b(\w*t|\w*d)\b/`, which looks for words ending in either `t` or `d`. Each time you use this pattern, the engine will look for a word boundary. It will then do the first alternation, looking for as many word characters in a row as possible. Then it looks for a `t`. Hmm—it won’t find one, because it already read all the word characters. So it will have to back up a character. If that character is a `t`, that’s great—now it can look for a word boundary, and then it’s all done. Otherwise, if there was no match, the engine keeps backing up and trying to find a `t`. If it runs all the way back to the initial word boundary, then the engine tries the second half of the alternation, looking for a `d` at the end.

You can see that this is a very complicated process. Well, the regular expression engine is meant to do complicated work, but this particular pattern makes that work much more complicated than it has to be.

An obvious shortcoming is that if the engine starts out at the beginning of a word that ends in `d`, it has to go all the way to the end and back searching fruitlessly for a `t` before it even starts looking for a `d`. We can definitely fix this. Let’s get rid of the alternation:

```
/\b\w*[td]\b/
```

This is an improvement. Now, the engine will scan the length of the word only once, regardless of whether it ends in `t`, `d`, or something else.

We still haven’t addressed the general backtracking issue. Notice that there is no need for the regular expression engine to continue backtracking more than a single character back from the end of a word. If that character isn't a t or d, there's no point in continuing, because even if we did find one earlier in the string it wouldn't be at the end of the word.

There's no way to force Perl to change this backtracking behavior (at least not so far as I know), but you can approach the problem in a slightly different manner. Ask yourself: "If I were looking for words ending in t or d, what would I be looking for?" More than likely, you'd be looking at the ends of words. You'd be looking for something like:

```
/[td]\b/
```

Now, this is interesting. This little regular expression does everything that the other two do, even though it may not be obvious at first. But think about it. To the left of the t or d there will be zero or more \w characters. We don't care what sort of \w characters they are; so, tautologically if you will, once we have a t or d to the left of a word boundary, we have a word ending in t or d.

Naturally, this little regular expression runs much faster than either of the two above—about twice as fast, more or less. Obviously there's not much backtracking, because the expression matches only a single character!

Use memory-free parentheses

If you are using parentheses for grouping alone, you won't need a copy of what the parentheses matched. You can save the time required to make the copy by using Perl's memory-free parentheses:

■ **Use memory-free parentheses (?:…) to speed up pattern matches.**

There's no point in memorizing the contents of the inner parentheses in this pattern, so if you want to save a little time, use memory-free parentheses.

`($host) = m/(\w+(\.\w+)*)/;`	*Find hostname-like thing (*foo.bar.com*) and put it into* $host.
`($host) = m/(\w+(?:\.\w+)*)/;`	*Same thing, but no memory for the inner parens.*

The time saved isn't generally all that great, and memory-free parentheses don't exactly improve readability. But sometimes, every little bit of speed helps!

See Item 16 for more about memory-free parentheses.

Benchmark your regular expressions

As with many other things in Perl, one of the best ways to determine how to make a pattern match run quickly is to write several alternative implementations and benchmark them.

Let's use the Benchmark module (see Item 37) to see how much of a difference those memory-free parentheses above really make:

■ **Time your regular expressions with Benchmark.**

```
use Benchmark;
@data = <>;

my $host;
timethese (100,
 { mem => q{
  for (@data) {
   ($host) = m/(\w+(\.\w+)+)/; }
  },

 memfree => q{
  for (@data) {
   ($host) = m/(\w+(?:\.\w+)+)/; }
  }
 }
);
```

Read some data. (I used 1,000 lines of an HTTP access log.)

The test code goes in an eval *string (see Item 54).*

Some more test code.

The results:

```
  Benchmark: timing 100 iterations of mem, memfree...
         mem: 12 secs (12.23 usr  0.00 sys = 12.23 cpu)
     memfree: 11 secs (10.64 usr  0.00 sys = 10.64 cpu)
```

Not bad: it takes about 15 percent longer to run the version without the memory-free parentheses.