NAME

perlretut - Perl regular expressions tutorial

DESCRIPTION

This page provides a basic tutorial on understanding, creating and using regular expressions in Perl. It serves as a complement to the reference page on regular expressions *perlre*. Regular expressions are an integral part of the `m//`, `s///`, `qr//` and `split` operators and so this tutorial also overlaps with *"Regexp Quote-Like Operators" in perlop* and *"split" in perlfunc*.

Perl is widely renowned for excellence in text processing, and regular expressions are one of the big factors behind this fame. Perl regular expressions display an efficiency and flexibility unknown in most other computer languages. Mastering even the basics of regular expressions will allow you to manipulate text with surprising ease.

What is a regular expression? A regular expression is simply a string that describes a pattern. Patterns are in common use these days; examples are the patterns typed into a search engine to find web pages and the patterns used to list files in a directory, e.g., `ls *.txt` or `dir *.*`. In Perl, the patterns described by regular expressions are used to search strings, extract desired parts of strings, and to do search and replace operations.

Regular expressions have the undeserved reputation of being abstract and difficult to understand. Regular expressions are constructed using simple concepts like conditionals and loops and are no more difficult to understand than the corresponding `if` conditionals and `while` loops in the Perl language itself. In fact, the main challenge in learning regular expressions is just getting used to the terse notation used to express these concepts.

This tutorial flattens the learning curve by discussing regular expression concepts, along with their notation, one at a time and with many examples. The first part of the tutorial will progress from the simplest word searches to the basic regular expression concepts. If you master the first part, you will have all the tools needed to solve about 98% of your needs. The second part of the tutorial is for those comfortable with the basics and hungry for more power tools. It discusses the more advanced regular expression operators and introduces the latest cutting edge innovations in 5.6.0.

A note: to save time, 'regular expression' is often abbreviated as regexp or regex. Regexp is a more natural abbreviation than regex, but is harder to pronounce. The Perl pod documentation is evenly split on regexp vs regex; in Perl, there is more than one way to abbreviate it. We'll use regexp in this tutorial.

Part 1: The basics

Simple word matching

The simplest regexp is simply a word, or more generally, a string of characters. A regexp consisting of a word matches any string that contains that word:

```
"Hello World" =~ /World/;  # matches
```

What is this perl statement all about? `"Hello World"` is a simple double quoted string. `World` is the regular expression and the `//` enclosing `/World/` tells perl to search a string for a match. The operator `=~` associates the string with the regexp match and produces a true value if the regexp matched, or false if the regexp did not match. In our case, `World` matches the second word in `"Hello World"`, so the expression is true. Expressions like this are useful in conditionals:

```
if ("Hello World" =~ /World/) {
    print "It matches\n";
}
else {
    print "It doesn't match\n";
}
```

There are useful variations on this theme. The sense of the match can be reversed by using `!~` operator:

```
if ("Hello World" !~ /World/) {
    print "It doesn't match\n";
}
else {
    print "It matches\n";
}
```

The literal string in the regexp can be replaced by a variable:

```
$greeting = "World";
if ("Hello World" =~ /$greeting/) {
    print "It matches\n";
}
else {
    print "It doesn't match\n";
}
```

If you're matching against the special default variable `$_`, the `$_ =~` part can be omitted:

```
$_ = "Hello World";
if (/World/) {
    print "It matches\n";
}
else {
    print "It doesn't match\n";
}
```

And finally, the `//` default delimiters for a match can be changed to arbitrary delimiters by putting an `'m'` out front:

```
"Hello World" =~ m!World!;   # matches, delimited by '!'
"Hello World" =~ m{World};   # matches, note the matching '{}'
"/usr/bin/perl" =~ m"/perl"; # matches after '/usr/bin',
                             # '/' becomes an ordinary char
```

`/World/`, `m!World!`, and `m{World}` all represent the same thing. When, e.g., `""` is used as a delimiter, the forward slash `'/'` becomes an ordinary character and can be used in a regexp without trouble.

Let's consider how different regexps would match `"Hello World"`:

```
"Hello World" =~ /world/;  # doesn't match
"Hello World" =~ /o W/;    # matches
"Hello World" =~ /oW/;     # doesn't match
"Hello World" =~ /World /; # doesn't match
```

The first regexp `world` doesn't match because regexps are case-sensitive. The second regexp matches because the substring `'o W'` occurs in the string `"Hello World"`. The space character `' '` is treated like any other character in a regexp and is needed to match in this case. The lack of a space character is the reason the third regexp `'oW'` doesn't match. The fourth regexp `'World '` doesn't match because there is a space at the end of the regexp, but not at the end of the string. The lesson here is that regexps must match a part of the string *exactly* in order for the statement to be true.

If a regexp matches in more than one place in the string, perl will always match at the earliest possible point in the string:

```
"Hello World" =~ /o/;       # matches 'o' in 'Hello'
"That hat is red" =~ /hat/; # matches 'hat' in 'That'
```

With respect to character matching, there are a few more points you need to know about. First of all, not all characters can be used 'as is' in a match. Some characters, called **metacharacters**, are reserved for use in regexp notation. The metacharacters are

```
{}[]()^$.|*+?\
```

The significance of each of these will be explained in the rest of the tutorial, but for now, it is important only to know that a metacharacter can be matched by putting a backslash before it:

```
"2+2=4" =~ /2+2/;    # doesn't match, + is a metacharacter
"2+2=4" =~ /2\+2/;   # matches, \+ is treated like an ordinary +
"The interval is [0,1)." =~ /[0,1)./     # is a syntax error!
"The interval is [0,1)." =~ /\[0,1\)\./  # matches
"/usr/bin/perl" =~ /\/usr\/bin\/perl/;  # matches
```

In the last regexp, the forward slash `'/'` is also backslashed, because it is used to delimit the regexp. This can lead to LTS (leaning toothpick syndrome), however, and it is often more readable to change delimiters.

```
"/usr/bin/perl" =~ m!/usr/bin/perl!;    # easier to read
```

The backslash character `'\'` is a metacharacter itself and needs to be backslashed:

```
'C:\WIN32' =~ /C:\\WIN/;   # matches
```

In addition to the metacharacters, there are some ASCII characters which don't have printable character equivalents and are instead represented by **escape sequences**. Common examples are `\t` for a tab, `\n` for a newline, `\r` for a carriage return and `\a` for a bell. If your string is better thought of as a sequence of arbitrary bytes, the octal escape sequence, e.g., `\033`, or hexadecimal escape sequence, e.g., `\x1B` may be a more natural representation for your bytes. Here are some examples of escapes:

```
"1000\t2000" =~ m(0\t2)   # matches
"1000\n2000" =~ /0\n20/   # matches
"1000\t2000" =~ /\000\t2/ # doesn't match, "0" ne "\000"
"cat"        =~ /\143\x61\x74/ # matches, but a weird way to spell cat
```

If you've been around Perl a while, all this talk of escape sequences may seem familiar. Similar escape sequences are used in double-quoted strings and in fact the regexps in Perl are mostly treated as double-quoted strings. This means that variables can be used in regexps as well. Just like double-quoted strings, the values of the variables in the regexp will be substituted in before the regexp is evaluated for matching purposes. So we have:

```
$foo = 'house';
'housecat' =~ /$foo/;      # matches
'cathouse' =~ /cat$foo/;   # matches
'housecat' =~ /${foo}cat/; # matches
```

So far, so good. With the knowledge above you can already perform searches with just about any literal string regexp you can dream up. Here is a *very simple* emulation of the Unix grep program:

```
% cat > simple_grep
#!/usr/bin/perl
$regexp = shift;
while (<>) {
    print if /$regexp/;
}
^D

% chmod +x simple_grep

% simple_grep abba /usr/dict/words
Babbage
cabbage
cabbages
sabbath
Sabbathize
Sabbathizes
sabbatical
scabbard
scabbards
```

This program is easy to understand. `#!/usr/bin/perl` is the standard way to invoke a perl program from the shell. `$regexp = shift;` saves the first command line argument as the regexp to be used, leaving the rest of the command line arguments to be treated as files. `while (<>)` loops over all the lines in all the files. For each line, `print if /$regexp/;` prints the line if the regexp matches the line. In this line, both `print` and `/$regexp/` use the default variable `$_` implicitly.

With all of the regexps above, if the regexp matched anywhere in the string, it was considered a match. Sometimes, however, we'd like to specify *where* in the string the regexp should try to match. To do this, we would use the **anchor** metacharacters ^ and $. The anchor ^ means match at the beginning of the string and the anchor $ means match at the end of the string, or before a newline at the end of the string. Here is how they are used:

```
"housekeeper" =~ /keeper/;    # matches
"housekeeper" =~ /^keeper/;   # doesn't match
"housekeeper" =~ /keeper$/;   # matches
"housekeeper\n" =~ /keeper$/; # matches
```

The second regexp doesn't match because ^ constrains `keeper` to match only at the beginning of the string, but `"housekeeper"` has keeper starting in the middle. The third regexp does match, since the $ constrains `keeper` to match only at the end of the string.

When both ^ and $ are used at the same time, the regexp has to match both the beginning and the end of the string, i.e., the regexp matches the whole string. Consider

```
"keeper" =~ /^keep$/;      # doesn't match
"keeper" =~ /^keeper$/;    # matches
""       =~ /^$/;          # ^$ matches an empty string
```

The first regexp doesn't match because the string has more to it than `keep`. Since the second regexp is exactly the string, it matches. Using both ^ and $ in a regexp forces the complete string to match, so it gives you complete control over which strings match and which don't. Suppose you are looking for a fellow named bert, off in a string by himself:

```
"dogbert" =~ /bert/;   # matches, but not what you want
```

```
"dilbert" =~ /^bert/;  # doesn't match, but ..
"bertram" =~ /^bert/;  # matches, so still not good enough

"bertram" =~ /^bert$/; # doesn't match, good
"dilbert" =~ /^bert$/; # doesn't match, good
"bert"    =~ /^bert$/; # matches, perfect
```

Of course, in the case of a literal string, one could just as easily use the string equivalence `$string eq 'bert'` and it would be more efficient. The `^...$` regexp really becomes useful when we add in the more powerful regexp tools below.

Using character classes

Although one can already do quite a lot with the literal string regexps above, we've only scratched the surface of regular expression technology. In this and subsequent sections we will introduce regexp concepts (and associated metacharacter notations) that will allow a regexp to not just represent a single character sequence, but a *whole class* of them.

One such concept is that of a **character class**. A character class allows a set of possible characters, rather than just a single character, to match at a particular point in a regexp. Character classes are denoted by brackets `[...]`, with the set of characters to be possibly matched inside. Here are some examples:

```
/cat/;       # matches 'cat'
/[bcr]at/;   # matches 'bat, 'cat', or 'rat'
/item[0123456789]/;  # matches 'item0' or ... or 'item9'
"abc" =~ /[cab]/;    # matches 'a'
```

In the last statement, even though `'c'` is the first character in the class, `'a'` matches because the first character position in the string is the earliest point at which the regexp can match.

```
/[yY][eE][sS]/;      # match 'yes' in a case-insensitive way
                     # 'yes', 'Yes', 'YES', etc.
```

This regexp displays a common task: perform a case-insensitive match. Perl provides away of avoiding all those brackets by simply appending an `'i'` to the end of the match. Then `/[yY][eE][sS]/;` can be rewritten as `/yes/i;`. The `'i'` stands for case-insensitive and is an example of a **modifier** of the matching operation. We will meet other modifiers later in the tutorial.

We saw in the section above that there were ordinary characters, which represented themselves, and special characters, which needed a backslash `\` to represent themselves. The same is true in a character class, but the sets of ordinary and special characters inside a character class are different than those outside a character class. The special characters for a character class are `-]\^$.` `]` is special because it denotes the end of a character class. `$` is special because it denotes a scalar variable. `\` is special because it is used in escape sequences, just like above. Here is how the special characters `]$\` are handled:

```
/[\]c]def/; # matches ']def' or 'cdef'
$x = 'bcr';
/[$x]at/;   # matches 'bat, 'cat', or 'rat'
/[\$x]at/;  # matches '$at' or 'xat'
/[\\$x]at/; # matches '\at', 'bat, 'cat', or 'rat'
```

The last two are a little tricky. in `[\$x]`, the backslash protects the dollar sign, so the character class has two members `$` and `x`. In `[\\$x]`, the backslash is protected, so `$x` is treated as a variable and substituted in double quote fashion.

The special character '-' acts as a range operator within character classes, so that a contiguous set of characters can be written as a range. With ranges, the unwieldy [0123456789] and [abc...xyz] become the svelte [0-9] and [a-z]. Some examples are

```
/item[0-9]/;  # matches 'item0' or ... or 'item9'
/[0-9bx-z]aa/;  # matches '0aa', ..., '9aa',
                # 'baa', 'xaa', 'yaa', or 'zaa'
/[0-9a-fA-F]/;  # matches a hexadecimal digit
/[0-9a-zA-Z_]/; # matches a "word" character,
                # like those in a perl variable name
```

If '-' is the first or last character in a character class, it is treated as an ordinary character; [-ab], [ab-] and [a\-b] are all equivalent.

The special character ^ in the first position of a character class denotes a **negated character class**, which matches any character but those in the brackets. Both [...] and [^...] must match a character, or the match fails. Then

```
/[^a]at/;  # doesn't match 'aat' or 'at', but matches
           # all other 'bat', 'cat, '0at', '%at', etc.
/[^0-9]/;  # matches a non-numeric character
/[a^]at/;  # matches 'aat' or '^at'; here '^' is ordinary
```

Now, even [0-9] can be a bother the write multiple times, so in the interest of saving keystrokes and making regexps more readable, Perl has several abbreviations for common character classes:

- \d is a digit and represents [0-9]
- \s is a whitespace character and represents [\ \t\r\n\f]
- \w is a word character (alphanumeric or _) and represents [0-9a-zA-Z_]
- \D is a negated \d; it represents any character but a digit [^0-9]
- \S is a negated \s; it represents any non-whitespace character [^\s]
- \W is a negated \w; it represents any non-word character [^\w]
- The period '.' matches any character but "\n"

The \d\s\w\D\S\W abbreviations can be used both inside and outside of character classes. Here are some in use:

```
/\d\d:\d\d:\d\d/; # matches a hh:mm:ss time format
/[\d\s]/;         # matches any digit or whitespace character
/\w\W\w/;         # matches a word char, followed by a
                  # non-word char, followed by a word char
/..rt/;           # matches any two chars, followed by 'rt'
/end\./;          # matches 'end.'
/end[.]/;         # same thing, matches 'end.'
```

Because a period is a metacharacter, it needs to be escaped to match as an ordinary period. Because, for example, \d and \w are sets of characters, it is incorrect to think of [^\d\w] as [\D\W]; in fact [^\d\w] is the same as [^\w], which is the same as [\W]. Think DeMorgan's laws.

An anchor useful in basic regexps is the **word anchor** \b. This matches a boundary between a word character and a non-word character \w\W or \W\w:

```
$x = "Housecat catenates house and cat";
$x =~ /cat/;    # matches cat in 'housecat'
```

```
$x =~ /\bcat/;  # matches cat in 'catenates'
$x =~ /cat\b/;  # matches cat in 'housecat'
$x =~ /\bcat\b/;  # matches 'cat' at end of string
```

Note in the last example, the end of the string is considered a word boundary.

You might wonder why `'.'` matches everything but `"\n"` - why not every character? The reason is that often one is matching against lines and would like to ignore the newline characters. For instance, while the string `"\n"` represents one line, we would like to think of as empty. Then

```
""   =~ /^$/;    # matches
"\n" =~ /^$/;    # matches, "\n" is ignored

""   =~ /./;      # doesn't match; it needs a char
""   =~ /^.$/;    # doesn't match; it needs a char
"\n" =~ /^.$/;    # doesn't match; it needs a char other than "\n"
"a"  =~ /^.$/;    # matches
"a\n"  =~ /^.$/;  # matches, ignores the "\n"
```

This behavior is convenient, because we usually want to ignore newlines when we count and match characters in a line. Sometimes, however, we want to keep track of newlines. We might even want ^ and $ to anchor at the beginning and end of lines within the string, rather than just the beginning and end of the string. Perl allows us to choose between ignoring and paying attention to newlines by using the `//s` and `//m` modifiers. `//s` and `//m` stand for single line and multi-line and they determine whether a string is to be treated as one continuous string, or as a set of lines. The two modifiers affect two aspects of how the regexp is interpreted: 1) how the `'.'` character class is defined, and 2) where the anchors ^ and $ are able to match. Here are the four possible combinations:

- no modifiers (//): Default behavior. `'.'` matches any character except `"\n"`. ^ matches only at the beginning of the string and $ matches only at the end or before a newline at the end.
- s modifier (//s): Treat string as a single long line. `'.'` matches any character, even `"\n"`. ^ matches only at the beginning of the string and $ matches only at the end or before a newline at the end.
- m modifier (//m): Treat string as a set of multiple lines. `'.'` matches any character except `"\n"`. ^ and $ are able to match at the start or end of *any* line within the string.
- both s and m modifiers (//sm): Treat string as a single long line, but detect multiple lines. `'.'` matches any character, even `"\n"`. ^ and $, however, are able to match at the start or end of *any* line within the string.

Here are examples of `//s` and `//m` in action:

```
$x = "There once was a girl\nWho programmed in Perl\n";

$x =~ /^Who/;   # doesn't match, "Who" not at start of string
$x =~ /^Who/s;  # doesn't match, "Who" not at start of string
$x =~ /^Who/m;  # matches, "Who" at start of second line
$x =~ /^Who/sm; # matches, "Who" at start of second line

$x =~ /girl.Who/;   # doesn't match, "." doesn't match "\n"
$x =~ /girl.Who/s;  # matches, "." matches "\n"
$x =~ /girl.Who/m;  # doesn't match, "." doesn't match "\n"
$x =~ /girl.Who/sm; # matches, "." matches "\n"
```

Most of the time, the default behavior is what is want, but `//s` and `//m` are occasionally very useful. If

`//m` is being used, the start of the string can still be matched with `\A` and the end of string can still be matched with the anchors `\Z` (matches both the end and the newline before, like $), and `\z` (matches only the end):

```
$x =~ /^Who/m;   # matches, "Who" at start of second line
$x =~ /\AWho/m;  # doesn't match, "Who" is not at start of string

$x =~ /girl$/m;  # matches, "girl" at end of first line
$x =~ /girl\Z/m; # doesn't match, "girl" is not at end of string

$x =~ /Perl\Z/m; # matches, "Perl" is at newline before end
$x =~ /Perl\z/m; # doesn't match, "Perl" is not at end of string
```

We now know how to create choices among classes of characters in a regexp. What about choices among words or character strings? Such choices are described in the next section.

Matching this or that

Sometimes we would like to our regexp to be able to match different possible words or character strings. This is accomplished by using the **alternation** metacharacter `|`. To match `dog` or `cat`, we form the regexp `dog|cat`. As before, perl will try to match the regexp at the earliest possible point in the string. At each character position, perl will first try to match the first alternative, `dog`. If `dog` doesn't match, perl will then try the next alternative, `cat`. If `cat` doesn't match either, then the match fails and perl moves to the next position in the string. Some examples:

```
"cats and dogs" =~ /cat|dog|bird/;  # matches "cat"
"cats and dogs" =~ /dog|cat|bird/;  # matches "cat"
```

Even though `dog` is the first alternative in the second regexp, `cat` is able to match earlier in the string.

```
"cats"          =~ /c|ca|cat|cats/; # matches "c"
"cats"          =~ /cats|cat|ca|c/; # matches "cats"
```

Here, all the alternatives match at the first string position, so the first alternative is the one that matches. If some of the alternatives are truncations of the others, put the longest ones first to give them a chance to match.

```
"cab" =~ /a|b|c/ # matches "c"
                 # /a|b|c/ == /[abc]/
```

The last example points out that character classes are like alternations of characters. At a given character position, the first alternative that allows the regexp match to succeed will be the one that matches.

Grouping things and hierarchical matching

Alternation allows a regexp to choose among alternatives, but by itself it unsatisfying. The reason is that each alternative is a whole regexp, but sometime we want alternatives for just part of a regexp. For instance, suppose we want to search for housecats or housekeepers. The regexp `housecat|housekeeper` fits the bill, but is inefficient because we had to type `house` twice. It would be nice to have parts of the regexp be constant, like `house`, and some parts have alternatives, like `cat|keeper`.

The **grouping** metacharacters `( )` solve this problem. Grouping allows parts of a regexp to be treated as a single unit. Parts of a regexp are grouped by enclosing them in parentheses. Thus we could solve the `housecat|housekeeper` by forming the regexp as `house(cat|keeper)`. The regexp `house(cat|keeper)` means match `house` followed by either `cat` or `keeper`. Some more

examples are

```
/(a|b)b/;    # matches 'ab' or 'bb'
/(ac|b)b/;   # matches 'acb' or 'bb'
/(^a|b)c/;   # matches 'ac' at start of string or 'bc' anywhere
/(a|[bc])d/; # matches 'ad', 'bd', or 'cd'

/house(cat|)/;  # matches either 'housecat' or 'house'
/house(cat(s|)|)/;  # matches either 'housecats' or 'housecat' or
                    # 'house'.  Note groups can be nested.

/(19|20|)\d\d/;  # match years 19xx, 20xx, or the Y2K problem, xx
"20" =~ /(19|20|)\d\d/;  # matches the null alternative '()\d\d',
                         # because '20\d\d' can't match
```

Alternations behave the same way in groups as out of them: at a given string position, the leftmost alternative that allows the regexp to match is taken. So in the last example at the first string position, `"20"` matches the second alternative, but there is nothing left over to match the next two digits `\d\d`. So perl moves on to the next alternative, which is the null alternative and that works, since `"20"` is two digits.

The process of trying one alternative, seeing if it matches, and moving on to the next alternative if it doesn't, is called **backtracking**. The term 'backtracking' comes from the idea that matching a regexp is like a walk in the woods. Successfully matching a regexp is like arriving at a destination. There are many possible trailheads, one for each string position, and each one is tried in order, left to right. From each trailhead there may be many paths, some of which get you there, and some which are dead ends. When you walk along a trail and hit a dead end, you have to backtrack along the trail to an earlier point to try another trail. If you hit your destination, you stop immediately and forget about trying all the other trails. You are persistent, and only if you have tried all the trails from all the trailheads and not arrived at your destination, do you declare failure. To be concrete, here is a step-by-step analysis of what perl does when it tries to match the regexp

```
"abcde" =~ /(abd|abc)(df|d|de)/;
```

0 Start with the first letter in the string 'a'.

1 Try the first alternative in the first group 'abd'.

2 Match 'a' followed by 'b'. So far so good.

3 'd' in the regexp doesn't match 'c' in the string - a dead end. So backtrack two characters and pick the second alternative in the first group 'abc'.

4 Match 'a' followed by 'b' followed by 'c'. We are on a roll and have satisfied the first group. Set $1 to 'abc'.

5 Move on to the second group and pick the first alternative 'df'.

6 Match the 'd'.

7 'f' in the regexp doesn't match 'e' in the string, so a dead end. Backtrack one character and pick the second alternative in the second group 'd'.

8 'd' matches. The second grouping is satisfied, so set $2 to 'd'.

9 We are at the end of the regexp, so we are done! We have matched 'abcd' out of the string "abcde".

There are a couple of things to note about this analysis. First, the third alternative in the second group

'de' also allows a match, but we stopped before we got to it - at a given character position, leftmost wins. Second, we were able to get a match at the first character position of the string 'a'. If there were no matches at the first position, perl would move to the second character position 'b' and attempt the match all over again. Only when all possible paths at all possible character positions have been exhausted does perl give up and declare `$string =~ /(abd|abc)(df|d|de)/;` to be false.

Even with all this work, regexp matching happens remarkably fast. To speed things up, during compilation stage, perl compiles the regexp into a compact sequence of opcodes that can often fit inside a processor cache. When the code is executed, these opcodes can then run at full throttle and search very quickly.

Extracting matches

The grouping metacharacters `( )` also serve another completely different function: they allow the extraction of the parts of a string that matched. This is very useful to find out what matched and for text processing in general. For each grouping, the part that matched inside goes into the special variables `$1`, `$2`, etc. They can be used just as ordinary variables:

```
    # extract hours, minutes, seconds
    if ($time =~ /(\d\d):(\d\d):(\d\d)/) {    # match hh:mm:ss format
 $hours = $1;
 $minutes = $2;
 $seconds = $3;
    }
```

Now, we know that in scalar context, `$time =~ /(\d\d):(\d\d):(\d\d)/` returns a true or false value. In list context, however, it returns the list of matched values `($1,$2,$3)`. So we could write the code more compactly as

```
    # extract hours, minutes, seconds
    ($hours, $minutes, $second) = ($time =~ /(\d\d):(\d\d):(\d\d)/);
```

If the groupings in a regexp are nested, `$1` gets the group with the leftmost opening parenthesis, `$2` the next opening parenthesis, etc. For example, here is a complex regexp and the matching variables indicated below it:

```
    /(ab(cd|ef)((gi)|j))/;
     1  2      34
```

so that if the regexp matched, e.g., `$2` would contain 'cd' or 'ef'. For convenience, perl sets `$+` to the string held by the highest numbered `$1`, `$2`, ... that got assigned (and, somewhat related, `$^N` to the value of the `$1`, `$2`, ... most-recently assigned; i.e. the `$1`, `$2`, ... associated with the rightmost closing parenthesis used in the match).

Closely associated with the matching variables `$1`, `$2`, ... are the **backreferences** `\1`, `\2`, ... Backreferences are simply matching variables that can be used *inside* a regexp. This is a really nice feature - what matches later in a regexp can depend on what matched earlier in the regexp. Suppose we wanted to look for doubled words in text, like 'the the'. The following regexp finds all 3-letter doubles with a space in between:

```
    /(\w\w\w)\s\1/;
```

The grouping assigns a value to \1, so that the same 3 letter sequence is used for both parts. Here are some words with repeated parts:

```
    % simple_grep '^(\w\w\w\w|\w\w\w|\w\w|\w)\1$' /usr/dict/words
    beriberi
    booboo
```

```
    coco
    mama
    murmur
    papa
```

The regexp has a single grouping which considers 4-letter combinations, then 3-letter combinations, etc. and uses `\1` to look for a repeat. Although `$1` and `\1` represent the same thing, care should be taken to use matched variables `$1`, `$2`, ... only outside a regexp and backreferences `\1`, `\2`, ... only inside a regexp; not doing so may lead to surprising and/or undefined results.

In addition to what was matched, Perl 5.6.0 also provides the positions of what was matched with the `@-` and `@+` arrays. `$-[0]` is the position of the start of the entire match and `$+[0]` is the position of the end. Similarly, `$-[n]` is the position of the start of the `$n` match and `$+[n]` is the position of the end. If `$n` is undefined, so are `$-[n]` and `$+[n]`. Then this code

```
    $x = "Mmm...donut, thought Homer";
    $x =~ /^(Mmm|Yech)\.\.\.(donut|peas)/; # matches
    foreach $expr (1..$#-) {
        print "Match $expr: '${$expr}' at position
($-[$expr],$+[$expr])\n";
    }
```

prints

```
    Match 1: 'Mmm' at position (0,3)
    Match 2: 'donut' at position (6,11)
```

Even if there are no groupings in a regexp, it is still possible to find out what exactly matched in a string. If you use them, perl will set `$`` to the part of the string before the match, will set `$&` to the part of the string that matched, and will set `$'` to the part of the string after the match. An example:

```
    $x = "the cat caught the mouse";
    $x =~ /cat/;  # $` = 'the ', $& = 'cat', $' = ' caught the mouse'
    $x =~ /the/;  # $` = '', $& = 'the', $' = ' cat caught the mouse'
```

In the second match, `$` = ''` because the regexp matched at the first character position in the string and stopped, it never saw the second 'the'. It is important to note that using `$`` and `$'` slows down regexp matching quite a bit, and `$&` slows it down to a lesser extent, because if they are used in one regexp in a program, they are generated for <all> regexps in the program. So if raw performance is a goal of your application, they should be avoided. If you need them, use `@-` and `@+` instead:

```
    $` is the same as substr( $x, 0, $-[0] )
    $& is the same as substr( $x, $-[0], $+[0]-$-[0] )
    $' is the same as substr( $x, $+[0] )
```

Matching repetitions

The examples in the previous section display an annoying weakness. We were only matching 3-letter words, or syllables of 4 letters or less. We'd like to be able to match words or syllables of any length, without writing out tedious alternatives like `\w\w\w\w|\w\w\w|\w\w|\w`.

This is exactly the problem the **quantifier** metacharacters `?`, `*`, `+`, and `{}` were created for. They allow us to determine the number of repeats of a portion of a regexp we consider to be a match. Quantifiers are put immediately after the character, character class, or grouping that we want to specify. They have the following meanings:

- a? = match 'a' 1 or 0 times
- a* = match 'a' 0 or more times, i.e., any number of times
- a+ = match 'a' 1 or more times, i.e., at least once
- a{n,m} = match at least n times, but not more than m times.
- a{n,} = match at least n or more times
- a{n} = match exactly n times

Here are some examples:

```
/[a-z]+\s+\d*/;  # match a lowercase word, at least some space, and
                 # any number of digits
/(\w+)\s+\1/;    # match doubled words of arbitrary length
/y(es)?/i;       # matches 'y', 'Y', or a case-insensitive 'yes'
$year =~ /\d{2,4}/;  # make sure year is at least 2 but not more
                     # than 4 digits
$year =~ /\d{4}|\d{2}/;    # better match; throw out 3 digit dates
$year =~ /\d{2}(\d{2})?/;  # same thing written differently. However,
                           # this produces $1 and the other does not.

% simple_grep '^(\w+)\1$' /usr/dict/words   # isn't this easier?
beriberi
booboo
coco
mama
murmur
papa
```

For all of these quantifiers, perl will try to match as much of the string as possible, while still allowing the regexp to succeed. Thus with `/a?.../`, perl will first try to match the regexp with the a present; if that fails, perl will try to match the regexp without the a present. For the quantifier `*`, we get the following:

```
$x = "the cat in the hat";
$x =~ /^(.*)(cat)(.*)$/; # matches,
                         # $1 = 'the '
                         # $2 = 'cat'
                         # $3 = ' in the hat'
```

Which is what we might expect, the match finds the only `cat` in the string and locks onto it. Consider, however, this regexp:

```
$x =~ /^(.*)(at)(.*)$/; # matches,
                        # $1 = 'the cat in the h'
                        # $2 = 'at'
                        # $3 = ''   (0 matches)
```

One might initially guess that perl would find the `at` in `cat` and stop there, but that wouldn't give the longest possible string to the first quantifier `.*`. Instead, the first quantifier `.*` grabs as much of the string as possible while still having the regexp match. In this example, that means having the `at` sequence with the final `at` in the string. The other important principle illustrated here is that when there are two or more elements in a regexp, the *leftmost* quantifier, if there is one, gets to grab as much the string as possible, leaving the rest of the regexp to fight over scraps. Thus in our example, the first quantifier `.*` grabs most of the string, while the second quantifier `.*` gets the empty string.

Quantifiers that grab as much of the string as possible are called **maximal match** or **greedy** quantifiers.

When a regexp can match a string in several different ways, we can use the principles above to predict which way the regexp will match:

- Principle 0: Taken as a whole, any regexp will be matched at the earliest possible position in the string.
- Principle 1: In an alternation a|b|c..., the leftmost alternative that allows a match for the whole regexp will be the one used.
- Principle 2: The maximal matching quantifiers ?, *, + and {n,m} will in general match as much of the string as possible while still allowing the whole regexp to match.
- Principle 3: If there are two or more elements in a regexp, the leftmost greedy quantifier, if any, will match as much of the string as possible while still allowing the whole regexp to match. The next leftmost greedy quantifier, if any, will try to match as much of the string remaining available to it as possible, while still allowing the whole regexp to match. And so on, until all the regexp elements are satisfied.

As we have seen above, Principle 0 overrides the others - the regexp will be matched as early as possible, with the other principles determining how the regexp matches at that earliest character position.

Here is an example of these principles in action:

```
$x = "The programming republic of Perl";
$x =~ /^(.+)(e|r)(.*)$/;  # matches,
                          # $1 = 'The programming republic of Pe'
                          # $2 = 'r'
                          # $3 = 'l'
```

This regexp matches at the earliest string position, 'T'. One might think that e, being leftmost in the alternation, would be matched, but r produces the longest string in the first quantifier.

```
$x =~ /(m{1,2})(.*)$/;  # matches,
                        # $1 = 'mm'
                        # $2 = 'ing republic of Perl'
```

Here, The earliest possible match is at the first 'm' in programming. m{1,2} is the first quantifier, so it gets to match a maximal mm.

```
$x =~ /.*(m{1,2})(.*)$/;  # matches,
                          # $1 = 'm'
                          # $2 = 'ing republic of Perl'
```

Here, the regexp matches at the start of the string. The first quantifier .* grabs as much as possible, leaving just a single 'm' for the second quantifier m{1,2}.

```
$x =~ /(.?)(m{1,2})(.*)$/;  # matches,
                            # $1 = 'a'
                            # $2 = 'mm'
                            # $3 = 'ing republic of Perl'
```

Here, .? eats its maximal one character at the earliest possible position in the string, 'a' in programming, leaving m{1,2} the opportunity to match both m's. Finally,

```
"aXXXb" =~ /(X*)/; # matches with $1 = ''
```

because it can match zero copies of 'X' at the beginning of the string. If you definitely want to match at least one 'X', use X+, not X*.

Sometimes greed is not good. At times, we would like quantifiers to match a *minimal* piece of string, rather than a maximal piece. For this purpose, Larry Wall created the **minimal match** or **non-greedy** quantifiers ??,*?, +?, and {}?. These are the usual quantifiers with a ? appended to them. They have the following meanings:

- a?? = match 'a' 0 or 1 times. Try 0 first, then 1.
- a*? = match 'a' 0 or more times, i.e., any number of times, but as few times as possible
- a+? = match 'a' 1 or more times, i.e., at least once, but as few times as possible
- a{n,m}? = match at least n times, not more than m times, as few times as possible
- a{n,}? = match at least n times, but as few times as possible
- a{n}? = match exactly n times. Because we match exactly n times, a{n}? is equivalent to a{n} and is just there for notational consistency.

Let's look at the example above, but with minimal quantifiers:

```
$x = "The programming republic of Perl";
$x =~ /^(.+?)(e|r)(.*)$/; # matches,
                          # $1 = 'Th'
                          # $2 = 'e'
                          # $3 = ' programming republic of Perl'
```

The minimal string that will allow both the start of the string ^ and the alternation to match is Th, with the alternation e|r matching e. The second quantifier .* is free to gobble up the rest of the string.

```
$x =~ /(m{1,2}?)(.*?)$/;  # matches,
                          # $1 = 'm'
                          # $2 = 'ming republic of Perl'
```

The first string position that this regexp can match is at the first 'm' in programming. At this position, the minimal m{1,2}? matches just one 'm'. Although the second quantifier .*? would prefer to match no characters, it is constrained by the end-of-string anchor $ to match the rest of the string.

```
$x =~ /(.*?)(m{1,2}?)(.*)$/;  # matches,
                              # $1 = 'The progra'
                              # $2 = 'm'
                              # $3 = 'ming republic of Perl'
```

In this regexp, you might expect the first minimal quantifier .*? to match the empty string, because it is not constrained by a ^ anchor to match the beginning of the word. Principle 0 applies here, however. Because it is possible for the whole regexp to match at the start of the string, it *will* match at the start of the string. Thus the first quantifier has to match everything up to the first m. The second minimal quantifier matches just one m and the third quantifier matches the rest of the string.

```
$x =~ /(.??)(m{1,2})(.*)$/;  # matches,
                             # $1 = 'a'
                             # $2 = 'mm'
                             # $3 = 'ing republic of Perl'
```

Just as in the previous regexp, the first quantifier .?? can match earliest at position 'a', so it does. The second quantifier is greedy, so it matches mm, and the third matches the rest of the string.

We can modify principle 3 above to take into account non-greedy quantifiers:

- Principle 3: If there are two or more elements in a regexp, the leftmost greedy (non-greedy) quantifier, if any, will match as much (little) of the string as possible while still allowing the whole regexp to match. The next leftmost greedy (non-greedy) quantifier, if any, will try to match as much (little) of the string remaining available to it as possible, while still allowing the whole regexp to match. And so on, until all the regexp elements are satisfied.

Just like alternation, quantifiers are also susceptible to backtracking. Here is a step-by-step analysis of the example

```
$x = "the cat in the hat";
$x =~ /^(.*)(at)(.*)$/; # matches,
                        # $1 = 'the cat in the h'
                        # $2 = 'at'
                        # $3 = ''   (0 matches)
```

0 Start with the first letter in the string 't'.

1 The first quantifier '.*' starts out by matching the whole string 'the cat in the hat'.

2 'a' in the regexp element 'at' doesn't match the end of the string. Backtrack one character.

3 'a' in the regexp element 'at' still doesn't match the last letter of the string 't', so backtrack one more character.

4 Now we can match the 'a' and the 't'.

5 Move on to the third element '.*'. Since we are at the end of the string and '.*' can match 0 times, assign it the empty string.

6 We are done!

Most of the time, all this moving forward and backtracking happens quickly and searching is fast. There are some pathological regexps, however, whose execution time exponentially grows with the size of the string. A typical structure that blows up in your face is of the form

```
/(a|b+)*/;
```

The problem is the nested indeterminate quantifiers. There are many different ways of partitioning a string of length n between the + and *: one repetition with b+ of length n, two repetitions with the first b+ length k and the second with length n-k, m repetitions whose bits add up to length n, etc. In fact there are an exponential number of ways to partition a string as a function of length. A regexp may get lucky and match early in the process, but if there is no match, perl will try *every* possibility before giving up. So be careful with nested *'s, `{n,m}`'s, and +'s. The book *Mastering regular expressions* by Jeffrey Friedl gives a wonderful discussion of this and other efficiency issues.

Building a regexp

At this point, we have all the basic regexp concepts covered, so let's give a more involved example of a regular expression. We will build a regexp that matches numbers.

The first task in building a regexp is to decide what we want to match and what we want to exclude. In our case, we want to match both integers and floating point numbers and we want to reject any string that isn't a number.

The next task is to break the problem down into smaller problems that are easily converted into a regexp.

The simplest case is integers. These consist of a sequence of digits, with an optional sign in front. The digits we can represent with `\d+` and the sign can be matched with `[+-]`. Thus the integer

regexp is

```
    /[+-]?\d+/;  # matches integers
```

A floating point number potentially has a sign, an integral part, a decimal point, a fractional part, and an exponent. One or more of these parts is optional, so we need to check out the different possibilities. Floating point numbers which are in proper form include 123., 0.345, .34, -1e6, and 25.4E-72. As with integers, the sign out front is completely optional and can be matched by `[+-]?`. We can see that if there is no exponent, floating point numbers must have a decimal point, otherwise they are integers. We might be tempted to model these with `\d*\.\d*`, but this would also match just a single decimal point, which is not a number. So the three cases of floating point number sans exponent are

```
/[+-]?\d+\./;  # 1., 321., etc.
/[+-]?\.\d+/;  # .1, .234, etc.
/[+-]?\d+\.\d+/;  # 1.0, 30.56, etc.
```

These can be combined into a single regexp with a three-way alternation:

```
/[+-]?(\d+\.\d+|\d+\.|\.\d+)/;  # floating point, no exponent
```

In this alternation, it is important to put `'\d+\.\d+'` before `'\d+\.'`. If `'\d+\.'` were first, the regexp would happily match that and ignore the fractional part of the number.

Now consider floating point numbers with exponents. The key observation here is that *both* integers and numbers with decimal points are allowed in front of an exponent. Then exponents, like the overall sign, are independent of whether we are matching numbers with or without decimal points, and can be 'decoupled' from the mantissa. The overall form of the regexp now becomes clear:

```
    /^(optional sign)(integer | f.p. mantissa)(optional exponent)$/;
```

The exponent is an `e` or `E`, followed by an integer. So the exponent regexp is

```
/[eE][+-]?\d+/;  # exponent
```

Putting all the parts together, we get a regexp that matches numbers:

```
/^[+-]?(\d+\.\d+|\d+\.|\.\d+|\d+)([eE][+-]?\d+)?$/;  # Ta da!
```

Long regexps like this may impress your friends, but can be hard to decipher. In complex situations like this, the `//x` modifier for a match is invaluable. It allows one to put nearly arbitrary whitespace and comments into a regexp without affecting their meaning. Using it, we can rewrite our 'extended' regexp in the more pleasing form

```
/^
   [+-]?         # first, match an optional sign
   (             # then match integers or f.p. mantissas:
       \d+\.\d+  # mantissa of the form a.b
      |\d+\.     # mantissa of the form a.
      |\.\d+     # mantissa of the form .b
      |\d+       # integer of the form a
   )
   ([eE][+-]?\d+)?  # finally, optionally match an exponent
$/x;
```

If whitespace is mostly irrelevant, how does one include space characters in an extended regexp?

The answer is to backslash it `'\ '` or put it in a character class `[ ]` . The same thing goes for pound signs, use `\#` or `[#]`. For instance, Perl allows a space between the sign and the mantissa/integer, and we could add this to our regexp as follows:

```
/^
   [+-]?\ *      # first, match an optional sign *and space*
   (             # then match integers or f.p. mantissas:
       \d+\.\d+  # mantissa of the form a.b
      |\d+\.     # mantissa of the form a.
      |\.\d+     # mantissa of the form .b
      |\d+       # integer of the form a
   )
   ([eE][+-]?\d+)?  # finally, optionally match an exponent
$/x;
```

In this form, it is easier to see a way to simplify the alternation. Alternatives 1, 2, and 4 all start with `\d+`, so it could be factored out:

```
/^
   [+-]?\ *      # first, match an optional sign
   (             # then match integers or f.p. mantissas:
       \d+       # start out with a ...
       (
           \.\d* # mantissa of the form a.b or a.
       )?        # ? takes care of integers of the form a
      |\.\d+     # mantissa of the form .b
   )
   ([eE][+-]?\d+)?  # finally, optionally match an exponent
$/x;
```

or written in the compact form,

```
 /^[+-]?\ *(\d+(\.\d*)?|\.\d+)([eE][+-]?\d+)?$/;
```

This is our final regexp. To recap, we built a regexp by

- specifying the task in detail,
- breaking down the problem into smaller parts,
- translating the small parts into regexps,
- combining the regexps,
- and optimizing the final combined regexp.

These are also the typical steps involved in writing a computer program. This makes perfect sense, because regular expressions are essentially programs written a little computer language that specifies patterns.

Using regular expressions in Perl

The last topic of Part 1 briefly covers how regexps are used in Perl programs. Where do they fit into Perl syntax?

We have already introduced the matching operator in its default `/regexp/` and arbitrary delimiter `m!regexp!` forms. We have used the binding operator `=~` and its negation `!~` to test for string matches. Associated with the matching operator, we have discussed the single line `//s`, multi-line `//m`, case-insensitive `//i` and extended `//x` modifiers.

There are a few more things you might want to know about matching operators. First, we pointed out earlier that variables in regexps are substituted before the regexp is evaluated:

```
$pattern = 'Seuss';
while (<>) {
    print if /$pattern/;
}
```

This will print any lines containing the word `Seuss`. It is not as efficient as it could be, however, because perl has to re-evaluate `$pattern` each time through the loop. If `$pattern` won't be changing over the lifetime of the script, we can add the `//o` modifier, which directs perl to only perform variable substitutions once:

```
#!/usr/bin/perl
#    Improved simple_grep
$regexp = shift;
while (<>) {
    print if /$regexp/o;  # a good deal faster
}
```

If you change `$pattern` after the first substitution happens, perl will ignore it. If you don't want any substitutions at all, use the special delimiter `m''`:

```
@pattern = ('Seuss');
while (<>) {
    print if m'@pattern';  # matches literal '@pattern', not 'Seuss'
}
```

`m''` acts like single quotes on a regexp; all other `m` delimiters act like double quotes. If the regexp evaluates to the empty string, the regexp in the *last successful match* is used instead. So we have

```
"dog" =~ /d/;  # 'd' matches
"dogbert =~ //;  # this matches the 'd' regexp used before
```

The final two modifiers `//g` and `//c` concern multiple matches. The modifier `//g` stands for global matching and allows the matching operator to match within a string as many times as possible. In scalar context, successive invocations against a string will have `//g` jump from match to match, keeping track of position in the string as it goes along. You can get or set the position with the `pos()` function.

The use of `//g` is shown in the following example. Suppose we have a string that consists of words separated by spaces. If we know how many words there are in advance, we could extract the words using groupings:

```
$x = "cat dog house"; # 3 words
$x =~ /^\s*(\w+)\s+(\w+)\s+(\w+)\s*$/; # matches,
                                       # $1 = 'cat'
                                       # $2 = 'dog'
                                       # $3 = 'house'
```

But what if we had an indeterminate number of words? This is the sort of task `//g` was made for. To extract all words, form the simple regexp `(\w+)` and loop over all matches with `/(\w+)/g`:

```
while ($x =~ /(\w+)/g) {
    print "Word is $1, ends at position ", pos $x, "\n";
}
```

prints

```
Word is cat, ends at position 3
Word is dog, ends at position 7
Word is house, ends at position 13
```

A failed match or changing the target string resets the position. If you don't want the position reset after failure to match, add the //c, as in /regexp/gc. The current position in the string is associated with the string, not the regexp. This means that different strings have different positions and their respective positions can be set or read independently.

In list context, //g returns a list of matched groupings, or if there are no groupings, a list of matches to the whole regexp. So if we wanted just the words, we could use

```
@words = ($x =~ /(\w+)/g);  # matches,
                            # $word[0] = 'cat'
                            # $word[1] = 'dog'
                            # $word[2] = 'house'
```

Closely associated with the //g modifier is the \G anchor. The \G anchor matches at the point where the previous //g match left off. \G allows us to easily do context-sensitive matching:

```
$metric = 1;  # use metric units
...
$x = <FILE>;  # read in measurement
$x =~ /^([+-]?\d+)\s*/g;  # get magnitude
$weight = $1;
if ($metric) { # error checking
    print "Units error!" unless $x =~ /\Gkg\./g;
}
else {
    print "Units error!" unless $x =~ /\Glbs\./g;
}
$x =~ /\G\s+(widget|sprocket)/g;  # continue processing
```

The combination of //g and \G allows us to process the string a bit at a time and use arbitrary Perl logic to decide what to do next. Currently, the \G anchor is only fully supported when used to anchor to the start of the pattern.

\G is also invaluable in processing fixed length records with regexps. Suppose we have a snippet of coding region DNA, encoded as base pair letters ATCGTTGAAT... and we want to find all the stop codons TGA. In a coding region, codons are 3-letter sequences, so we can think of the DNA snippet as a sequence of 3-letter records. The naive regexp

```
# expanded, this is "ATC GTT GAA TGC AAA TGA CAT GAC"
$dna = "ATCGTTGAATGCAAATGACATGAC";
$dna =~ /TGA/;
```

doesn't work; it may match a TGA, but there is no guarantee that the match is aligned with codon boundaries, e.g., the substring GTT GAA gives a match. A better solution is

```
while ($dna =~ /(\w\w\w)*?TGA/g) {  # note the minimal *?
    print "Got a TGA stop codon at position ", pos $dna, "\n";
}
```

which prints

```
Got a TGA stop codon at position 18
Got a TGA stop codon at position 23
```

Position 18 is good, but position 23 is bogus. What happened?

The answer is that our regexp works well until we get past the last real match. Then the regexp will fail to match a synchronized TGA and start stepping ahead one character position at a time, not what we want. The solution is to use `\G` to anchor the match to the codon alignment:

```
while ($dna =~ /\G(\w\w\w)*?TGA/g) {
    print "Got a TGA stop codon at position ", pos $dna, "\n";
}
```

This prints

```
Got a TGA stop codon at position 18
```

which is the correct answer. This example illustrates that it is important not only to match what is desired, but to reject what is not desired.

search and replace

Regular expressions also play a big role in **search and replace** operations in Perl. Search and replace is accomplished with the `s///` operator. The general form is `s/regexp/replacement/modifiers`, with everything we know about regexps and modifiers applying in this case as well. The `replacement` is a Perl double quoted string that replaces in the string whatever is matched with the `regexp`. The operator =~ is also used here to associate a string with `s///`. If matching against `$_`, the `$_ =~` can be dropped. If there is a match, `s///` returns the number of substitutions made, otherwise it returns false. Here are a few examples:

```
$x = "Time to feed the cat!";
$x =~ s/cat/hacker/;   # $x contains "Time to feed the hacker!"
if ($x =~ s/^(Time.*hacker)!$/$1 now!/) {
    $more_insistent = 1;
}
$y = "'quoted words'";
$y =~ s/^'(.*)'$/$1/;  # strip single quotes,
                       # $y contains "quoted words"
```

In the last example, the whole string was matched, but only the part inside the single quotes was grouped. With the `s///` operator, the matched variables `$1`, `$2`, etc. are immediately available for use in the replacement expression, so we use `$1` to replace the quoted string with just what was quoted. With the global modifier, `s///g` will search and replace all occurrences of the regexp in the string:

```
$x = "I batted 4 for 4";
$x =~ s/4/four/;   # doesn't do it all:
                   # $x contains "I batted four for 4"
$x = "I batted 4 for 4";
$x =~ s/4/four/g;  # does it all:
                   # $x contains "I batted four for four"
```

If you prefer 'regex' over 'regexp' in this tutorial, you could use the following program to replace it:

```
% cat > simple_replace
#!/usr/bin/perl
$regexp = shift;
```

```
$replacement = shift;
while (<>) {
    s/$regexp/$replacement/go;
    print;
}
^D

% simple_replace regexp regex perlretut.pod
```

In `simple_replace` we used the `s///g` modifier to replace all occurrences of the regexp on each line and the `s///o` modifier to compile the regexp only once. As with `simple_grep`, both the `print` and the `s/$regexp/$replacement/go` use `$_` implicitly.

A modifier available specifically to search and replace is the `s///e` evaluation modifier. `s///e` wraps an `eval{...}` around the replacement string and the evaluated result is substituted for the matched substring. `s///e` is useful if you need to do a bit of computation in the process of replacing text. This example counts character frequencies in a line:

```
$x = "Bill the cat";
$x =~ s/(.)/$chars{$1}++;$1/eg;  # final $1 replaces char with itself
print "frequency of '$_' is $chars{$_}\n"
    foreach (sort {$chars{$b} <=> $chars{$a}} keys %chars);
```

This prints

```
frequency of ' ' is 2
frequency of 't' is 2
frequency of 'l' is 2
frequency of 'B' is 1
frequency of 'c' is 1
frequency of 'e' is 1
frequency of 'h' is 1
frequency of 'i' is 1
frequency of 'a' is 1
```

As with the match `m//` operator, `s///` can use other delimiters, such as `s!!!` and `s{}{}`, and even `s{}//`. If single quotes are used `s'''`, then the regexp and replacement are treated as single quoted strings and there are no substitutions. `s///` in list context returns the same thing as in scalar context, i.e., the number of matches.

The split operator

The **`split`** function can also optionally use a matching operator `m//` to split a string. `split /regexp/, string, limit` splits `string` into a list of substrings and returns that list. The regexp is used to match the character sequence that the `string` is split with respect to. The `limit`, if present, constrains splitting into no more than `limit` number of strings. For example, to split a string into words, use

```
$x = "Calvin and Hobbes";
@words = split /\s+/, $x;  # $word[0] = 'Calvin'
                           # $word[1] = 'and'
                           # $word[2] = 'Hobbes'
```

If the empty regexp `//` is used, the regexp always matches and the string is split into individual characters. If the regexp has groupings, then list produced contains the matched substrings from the groupings as well. For instance,

```
$x = "/usr/bin/perl";
@dirs = split m!/!, $x;  # $dirs[0] = ''
                         # $dirs[1] = 'usr'
                         # $dirs[2] = 'bin'
                         # $dirs[3] = 'perl'
@parts = split m!(/)!, $x;  # $parts[0] = ''
                            # $parts[1] = '/'
                            # $parts[2] = 'usr'
                            # $parts[3] = '/'
                            # $parts[4] = 'bin'
                            # $parts[5] = '/'
                            # $parts[6] = 'perl'
```

Since the first character of $x matched the regexp, `split` prepended an empty initial element to the list.

If you have read this far, congratulations! You now have all the basic tools needed to use regular expressions to solve a wide range of text processing problems. If this is your first time through the tutorial, why not stop here and play around with regexps a while... Part 2 concerns the more esoteric aspects of regular expressions and those concepts certainly aren't needed right at the start.

Part 2: Power tools

OK, you know the basics of regexps and you want to know more. If matching regular expressions is analogous to a walk in the woods, then the tools discussed in Part 1 are analogous to topo maps and a compass, basic tools we use all the time. Most of the tools in part 2 are analogous to flare guns and satellite phones. They aren't used too often on a hike, but when we are stuck, they can be invaluable.

What follows are the more advanced, less used, or sometimes esoteric capabilities of perl regexps. In Part 2, we will assume you are comfortable with the basics and concentrate on the new features.

More on characters, strings, and character classes

There are a number of escape sequences and character classes that we haven't covered yet.

There are several escape sequences that convert characters or strings between upper and lower case. `\l` and `\u` convert the next character to lower or upper case, respectively:

```
$x = "perl";
$string =~ /\u$x/;  # matches 'Perl' in $string
$x = "M(rs?|s)\\.";# note the double backslash
$string =~ /\l$x/;  # matches 'mr.', 'mrs.', and 'ms.',
```

`\L` and `\U` converts a whole substring, delimited by `\L` or `\U` and `\E`, to lower or upper case:

```
$x = "This word is in lower case:\L SHOUT\E";
$x =~ /shout/;       # matches
$x = "I STILL KEYPUNCH CARDS FOR MY 360"
$x =~ /\Ukeypunch/;  # matches punch card string
```

If there is no `\E`, case is converted until the end of the string. The regexps `\L\u$word` or `\u\L$word` convert the first character of `$word` to uppercase and the rest of the characters to lowercase.

Control characters can be escaped with `\c`, so that a control-Z character would be matched with `\cZ`. The escape sequence `\Q`...`\E` quotes, or protects most non-alphabetic characters. For instance,

```
$x = "\QThat !^*&%~& cat!";
$x =~ /\Q!^*&%~&\E/;  # check for rough language
```

It does not protect $ or @, so that variables can still be substituted.

With the advent of 5.6.0, perl regexps can handle more than just the standard ASCII character set. Perl now supports **Unicode**, a standard for encoding the character sets from many of the world's written languages. Unicode does this by allowing characters to be more than one byte wide. Perl uses the UTF-8 encoding, in which ASCII characters are still encoded as one byte, but characters greater than `chr(127)` may be stored as two or more bytes.

What does this mean for regexps? Well, regexp users don't need to know much about perl's internal representation of strings. But they do need to know 1) how to represent Unicode characters in a regexp and 2) when a matching operation will treat the string to be searched as a sequence of bytes (the old way) or as a sequence of Unicode characters (the new way). The answer to 1) is that Unicode characters greater than `chr(127)` may be represented using the `\x{hex}` notation, with `hex` a hexadecimal integer:

```
/\x{263a}/;  # match a Unicode smiley face :)
```

Unicode characters in the range of 128-255 use two hexadecimal digits with braces: `\x{ab}`. Note that this is different than `\xab`, which is just a hexadecimal byte with no Unicode significance.

NOTE: in Perl 5.6.0 it used to be that one needed to say `use utf8` to use any Unicode features. This is no more the case: for almost all Unicode processing, the explicit `utf8` pragma is not needed. (The only case where it matters is if your Perl script is in Unicode and encoded in UTF-8, then an explicit `use utf8` is needed.)

Figuring out the hexadecimal sequence of a Unicode character you want or deciphering someone else's hexadecimal Unicode regexp is about as much fun as programming in machine code. So another way to specify Unicode characters is to use the **named character** escape sequence `\N{name}`. `name` is a name for the Unicode character, as specified in the Unicode standard. For instance, if we wanted to represent or match the astrological sign for the planet Mercury, we could use

```
use charnames ":full"; # use named chars with Unicode full names
$x = "abc\N{MERCURY}def";
$x =~ /\N{MERCURY}/;   # matches
```

One can also use short names or restrict names to a certain alphabet:

```
use charnames ’:full’;
print "\N{GREEK SMALL LETTER SIGMA} is called sigma.\n";

use charnames ":short";
print "\N{greek:Sigma} is an upper-case sigma.\n";

use charnames qw(greek);
print "\N{sigma} is Greek sigma\n";
```

A list of full names is found in the file Names.txt in the lib/perl5/5.X.X/unicore directory.

The answer to requirement 2), as of 5.6.0, is that if a regexp contains Unicode characters, the string is searched as a sequence of Unicode characters. Otherwise, the string is searched as a sequence of bytes. If the string is being searched as a sequence of Unicode characters, but matching a single byte is required, we can use the `\C` escape sequence. `\C` is a character class akin to `.` except that it matches *any* byte 0-255. So

```
use charnames ":full"; # use named chars with Unicode full names
$x = "a";
```

```
    $x =~ /\C/;  # matches 'a', eats one byte
    $x = "";
    $x =~ /\C/;  # doesn't match, no bytes to match
    $x = "\N{MERCURY}";  # two-byte Unicode character
    $x =~ /\C/;  # matches, but dangerous!
```

The last regexp matches, but is dangerous because the string *character* position is no longer synchronized to the string *byte* position. This generates the warning 'Malformed UTF-8 character'. The `\C` is best used for matching the binary data in strings with binary data intermixed with Unicode characters.

Let us now discuss the rest of the character classes. Just as with Unicode characters, there are named Unicode character classes represented by the `\p{name}` escape sequence. Closely associated is the `\P{name}` character class, which is the negation of the `\p{name}` class. For example, to match lower and uppercase characters,

```
    use charnames ":full"; # use named chars with Unicode full names
    $x = "BOB";
    $x =~ /^\p{IsUpper}/;   # matches, uppercase char class
    $x =~ /^\P{IsUpper}/;   # doesn't match, char class sans uppercase
    $x =~ /^\p{IsLower}/;   # doesn't match, lowercase char class
    $x =~ /^\P{IsLower}/;   # matches, char class sans lowercase
```

Here is the association between some Perl named classes and the traditional Unicode classes:

```
    Perl class name  Unicode class name or regular expression

    IsAlpha          /^[LM]/
    IsAlnum          /^[LMN]/
    IsASCII          $code <= 127
    IsCntrl          /^C/
    IsBlank          $code =~ /^(0020|0009)$/ || /^Z[^lp]/
    IsDigit          Nd
    IsGraph          /^([LMNPS]|Co)/
    IsLower          Ll
    IsPrint          /^([LMNPS]|Co|Zs)/
    IsPunct          /^P/
    IsSpace          /^Z/ || ($code =~ /^(0009|000A|000B|000C|000D)$/
    IsSpacePerl      /^Z/ || ($code =~
/^(0009|000A|000C|000D|0085|2028|2029)$/
    IsUpper          /^L[ut]/
    IsWord           /^[LMN]/ || $code eq "005F"
    IsXDigit         $code =~ /^00(3[0-9]|[46][1-6])$/
```

You can also use the official Unicode class names with the `\p` and `\P`, like `\p{L}` for Unicode 'letters', or `\p{Lu}` for uppercase letters, or `\P{Nd}` for non-digits. If a `name` is just one letter, the braces can be dropped. For instance, `\pM` is the character class of Unicode 'marks', for example accent marks. For the full list see *perlunicode*.

The Unicode has also been separated into various sets of characters which you can test with `\p{In...}` (in) and `\P{In...}` (not in), for example `\p{Latin}`, `\p{Greek}`, or `\P{Katakana}`. For the full list see *perlunicode*.

`\X` is an abbreviation for a character class sequence that includes the Unicode 'combining character sequences'. A 'combining character sequence' is a base character followed by any number of combining characters. An example of a combining character is an accent. Using the Unicode full

names, e.g., A + COMBINING RING is a combining character sequence with base character A and combining character COMBINING RING , which translates in Danish to A with the circle atop it, as in the word Angstrom. \X is equivalent to \PM\pM*}, i.e., a non-mark followed by one or more marks.

For the full and latest information about Unicode see the latest Unicode standard, or the Unicode Consortium's website http://www.unicode.org/

As if all those classes weren't enough, Perl also defines POSIX style character classes. These have the form [:name:], with name the name of the POSIX class. The POSIX classes are alpha, alnum, ascii, cntrl, digit, graph, lower, print, punct, space, upper, and xdigit, and two extensions, word (a Perl extension to match \w), and blank (a GNU extension). If utf8 is being used, then these classes are defined the same as their corresponding perl Unicode classes: [:upper:] is the same as \p{IsUpper}, etc. The POSIX character classes, however, don't require using utf8. The [:digit:], [:word:], and [:space:] correspond to the familiar \d, \w, and \s character classes. To negate a POSIX class, put a ^ in front of the name, so that, e.g., [:^digit:] corresponds to \D and under utf8, \P{IsDigit}. The Unicode and POSIX character classes can be used just like \d, with the exception that POSIX character classes can only be used inside of a character class:

```
/\s+[abc[:digit:]xyz]\s*/;  # match a,b,c,x,y,z, or a digit
/^=item\s[[:digit:]]/;      # match '=item',
                            # followed by a space and a digit
use charnames ":full";
/\s+[abc\p{IsDigit}xyz]\s+/;  # match a,b,c,x,y,z, or a digit
/^=item\s\p{IsDigit}/;        # match '=item',
                              # followed by a space and a digit
```

Whew! That is all the rest of the characters and character classes.

Compiling and saving regular expressions

In Part 1 we discussed the //o modifier, which compiles a regexp just once. This suggests that a compiled regexp is some data structure that can be stored once and used again and again. The regexp quote qr// does exactly that: qr/string/ compiles the string as a regexp and transforms the result into a form that can be assigned to a variable:

```
$reg = qr/foo+bar?/;  # reg contains a compiled regexp
```

Then $reg can be used as a regexp:

```
$x = "fooooba";
$x =~ $reg;     # matches, just like /foo+bar?/
$x =~ /$reg/;   # same thing, alternate form
```

$reg can also be interpolated into a larger regexp:

```
$x =~ /(abc)?$reg/;  # still matches
```

As with the matching operator, the regexp quote can use different delimiters, e.g., qr!!, qr{} and qr~~. The single quote delimiters qr'' prevent any interpolation from taking place.

Pre-compiled regexps are useful for creating dynamic matches that don't need to be recompiled each time they are encountered. Using pre-compiled regexps, simple_grep program can be expanded into a program that matches multiple patterns:

```
% cat > multi_grep
#!/usr/bin/perl
# multi_grep - match any of <number> regexps
```

```
# usage: multi_grep <number> regexp1 regexp2 ... file1 file2 ...

$number = shift;
$regexp[$_] = shift foreach (0..$number-1);
@compiled = map qr/$_/, @regexp;
while ($line = <>) {
    foreach $pattern (@compiled) {
        if ($line =~ /$pattern/) {
            print $line;
            last;  # we matched, so move onto the next line
        }
    }
}
^D

% multi_grep 2 last for multi_grep
    $regexp[$_] = shift foreach (0..$number-1);
        foreach $pattern (@compiled) {
                last;
```

Storing pre-compiled regexps in an array `@compiled` allows us to simply loop through the regexps without any recompilation, thus gaining flexibility without sacrificing speed.

Embedding comments and modifiers in a regular expression

Starting with this section, we will be discussing Perl's set of **extended patterns**. These are extensions to the traditional regular expression syntax that provide powerful new tools for pattern matching. We have already seen extensions in the form of the minimal matching constructs `??`, `*?`, `+?`, `{n,m}?`, and `{n,}?`. The rest of the extensions below have the form `(?char...)`, where the `char` is a character that determines the type of extension.

The first extension is an embedded comment `(?#text)`. This embeds a comment into the regular expression without affecting its meaning. The comment should not have any closing parentheses in the text. An example is

```
/(?# Match an integer:)[+-]?\d+/;
```

This style of commenting has been largely superseded by the raw, freeform commenting that is allowed with the `//x` modifier.

The modifiers `//i`, `//m`, `//s`, and `//x` can also embedded in a regexp using `(?i)`, `(?m)`, `(?s)`, and `(?x)`. For instance,

```
/(?i)yes/;  # match 'yes' case insensitively
/yes/i;     # same thing
/(?x)(          # freeform version of an integer regexp
         [+-]?  # match an optional sign
         \d+    # match a sequence of digits
     )
/x;
```

Embedded modifiers can have two important advantages over the usual modifiers. Embedded modifiers allow a custom set of modifiers to *each* regexp pattern. This is great for matching an array of regexps that must have different modifiers:

```
$pattern[0] = '(?i)doctor';
$pattern[1] = 'Johnson';
```

```
...
while (<>) {
    foreach $patt (@pattern) {
        print if /$patt/;
    }
}
```

The second advantage is that embedded modifiers only affect the regexp inside the group the embedded modifier is contained in. So grouping can be used to localize the modifier's effects:

```
/Answer: ((?i)yes)/;  # matches 'Answer: yes', 'Answer: YES', etc.
```

Embedded modifiers can also turn off any modifiers already present by using, e.g., `(?-i)`. Modifiers can also be combined into a single expression, e.g., `(?s-i)` turns on single line mode and turns off case insensitivity.

Non-capturing groupings

We noted in Part 1 that groupings `()` had two distinct functions: 1) group regexp elements together as a single unit, and 2) extract, or capture, substrings that matched the regexp in the grouping. Non-capturing groupings, denoted by `(?:regexp)`, allow the regexp to be treated as a single unit, but don't extract substrings or set matching variables `$1`, etc. Both capturing and non-capturing groupings are allowed to co-exist in the same regexp. Because there is no extraction, non-capturing groupings are faster than capturing groupings. Non-capturing groupings are also handy for choosing exactly which parts of a regexp are to be extracted to matching variables:

```
# match a number, $1-$4 are set, but we only want $1
/([+-]?\ *(\d+(\.\d*)?|\.\d+)([eE][+-]?\d+)?)/;

# match a number faster , only $1 is set
/([+-]?\ *(?:\d+(?:\.\d*)?|\.\d+)(?:[eE][+-]?\d+)?)/;

# match a number, get $1 = whole number, $2 = exponent
/([+-]?\ *(?:\d+(?:\.\d*)?|\.\d+)(?:[eE]([+-]?\d+))?)/;
```

Non-capturing groupings are also useful for removing nuisance elements gathered from a split operation:

```
$x = '12a34b5';
@num = split /(a|b)/, $x;    # @num = ('12','a','34','b','5')
@num = split /(?:a|b)/, $x;  # @num = ('12','34','5')
```

Non-capturing groupings may also have embedded modifiers: `(?i-m:regexp)` is a non-capturing grouping that matches `regexp` case insensitively and turns off multi-line mode.

Looking ahead and looking behind

This section concerns the lookahead and lookbehind assertions. First, a little background.

In Perl regular expressions, most regexp elements 'eat up' a certain amount of string when they match. For instance, the regexp element `[abc}]` eats up one character of the string when it matches, in the sense that perl moves to the next character position in the string after the match. There are some elements, however, that don't eat up characters (advance the character position) if they match. The examples we have seen so far are the anchors. The anchor `^` matches the beginning of the line, but doesn't eat any characters. Similarly, the word boundary anchor `\b` matches, e.g., if the character to the left is a word character and the character to the right is a non-word character, but it doesn't eat up any characters itself. Anchors are examples of 'zero-width assertions'. Zero-width, because they

consume no characters, and assertions, because they test some property of the string. In the context of our walk in the woods analogy to regexp matching, most regexp elements move us along a trail, but anchors have us stop a moment and check our surroundings. If the local environment checks out, we can proceed forward. But if the local environment doesn't satisfy us, we must backtrack.

Checking the environment entails either looking ahead on the trail, looking behind, or both. ^ looks behind, to see that there are no characters before. $ looks ahead, to see that there are no characters after. \b looks both ahead and behind, to see if the characters on either side differ in their 'word'-ness.

The lookahead and lookbehind assertions are generalizations of the anchor concept. Lookahead and lookbehind are zero-width assertions that let us specify which characters we want to test for. The lookahead assertion is denoted by `(?=regexp)` and the lookbehind assertion is denoted by `(?<=fixed-regexp)`. Some examples are

```
$x = "I catch the housecat 'Tom-cat' with catnip";
$x =~ /cat(?=\s+)/;  # matches 'cat' in 'housecat'
@catwords = ($x =~ /(?<=\s)cat\w+/g);  # matches,
                                       # $catwords[0] = 'catch'
                                       # $catwords[1] = 'catnip'
$x =~ /\bcat\b/;  # matches 'cat' in 'Tom-cat'
$x =~ /(?<=\s)cat(?=\s)/; # doesn't match; no isolated 'cat' in
                          # middle of $x
```

Note that the parentheses in `(?=regexp)` and `(?<=regexp)` are non-capturing, since these are zero-width assertions. Thus in the second regexp, the substrings captured are those of the whole regexp itself. Lookahead `(?=regexp)` can match arbitrary regexps, but lookbehind `(?<=fixed-regexp)` only works for regexps of fixed width, i.e., a fixed number of characters long. Thus `(?<=(ab|bc))` is fine, but `(?<=(ab)*)` is not. The negated versions of the lookahead and lookbehind assertions are denoted by `(?!regexp)` and `(?<!fixed-regexp)` respectively. They evaluate true if the regexps do *not* match:

```
$x = "foobar";
$x =~ /foo(?!bar)/;  # doesn't match, 'bar' follows 'foo'
$x =~ /foo(?!baz)/;  # matches, 'baz' doesn't follow 'foo'
$x =~ /(?<!\s)foo/;  # matches, there is no \s before 'foo'
```

The `\C` is unsupported in lookbehind, because the already treacherous definition of `\C` would become even more so when going backwards.

Using independent subexpressions to prevent backtracking

The last few extended patterns in this tutorial are experimental as of 5.6.0. Play with them, use them in some code, but don't rely on them just yet for production code.

Independent subexpressions are regular expressions, in the context of a larger regular expression, that function independently of the larger regular expression. That is, they consume as much or as little of the string as they wish without regard for the ability of the larger regexp to match. Independent subexpressions are represented by `(?>regexp)`. We can illustrate their behavior by first considering an ordinary regexp:

```
$x = "ab";
$x =~ /a*ab/;  # matches
```

This obviously matches, but in the process of matching, the subexpression `a*` first grabbed the a. Doing so, however, wouldn't allow the whole regexp to match, so after backtracking, `a*` eventually gave back the a and matched the empty string. Here, what `a*` matched was *dependent* on what the rest of the regexp matched.

Contrast that with an independent subexpression:

```
$x =~ /(?>a*)ab/;  # doesn't match!
```

The independent subexpression `(?>a*)` doesn't care about the rest of the regexp, so it sees an a and grabs it. Then the rest of the regexp `ab` cannot match. Because `(?>a*)` is independent, there is no backtracking and the independent subexpression does not give up its a. Thus the match of the regexp as a whole fails. A similar behavior occurs with completely independent regexps:

```
$x = "ab";
$x =~ /a*/g;   # matches, eats an 'a'
$x =~ /\Gab/g; # doesn't match, no 'a' available
```

Here `//g` and `\G` create a 'tag team' handoff of the string from one regexp to the other. Regexps with an independent subexpression are much like this, with a handoff of the string to the independent subexpression, and a handoff of the string back to the enclosing regexp.

The ability of an independent subexpression to prevent backtracking can be quite useful. Suppose we want to match a non-empty string enclosed in parentheses up to two levels deep. Then the following regexp matches:

```
$x = "abc(de(fg)h";  # unbalanced parentheses
$x =~ /\( ( [^()]+ | \([^()]*\) )+ \)/x;
```

The regexp matches an open parenthesis, one or more copies of an alternation, and a close parenthesis. The alternation is two-way, with the first alternative `[^()]+` matching a substring with no parentheses and the second alternative `\([^()]*\)` matching a substring delimited by parentheses. The problem with this regexp is that it is pathological: it has nested indeterminate quantifiers of the form `(a+|b)+`. We discussed in Part 1 how nested quantifiers like this could take an exponentially long time to execute if there was no match possible. To prevent the exponential blowup, we need to prevent useless backtracking at some point. This can be done by enclosing the inner quantifier as an independent subexpression:

```
$x =~ /\( ( (?>[^()]+) | \([^()]*\) )+ \)/x;
```

Here, `(?>[^()]+)` breaks the degeneracy of string partitioning by gobbling up as much of the string as possible and keeping it. Then match failures fail much more quickly.

Conditional expressions

A **conditional expression** is a form of if-then-else statement that allows one to choose which patterns are to be matched, based on some condition. There are two types of conditional expression: `(?(condition)yes-regexp)` and `(?(condition)yes-regexp|no-regexp)`. `(?(condition)yes-regexp)` is like an `'if () {}'` statement in Perl. If the `condition` is true, the `yes-regexp` will be matched. If the `condition` is false, the `yes-regexp` will be skipped and perl will move onto the next regexp element. The second form is like an `'if () {} else {}'` statement in Perl. If the `condition` is true, the `yes-regexp` will be matched, otherwise the `no-regexp` will be matched.

The `condition` can have two forms. The first form is simply an integer in parentheses `(integer)`. It is true if the corresponding backreference `\integer` matched earlier in the regexp. The second form is a bare zero width assertion `(?...)`, either a lookahead, a lookbehind, or a code assertion (discussed in the next section).

The integer form of the `condition` allows us to choose, with more flexibility, what to match based on what matched earlier in the regexp. This searches for words of the form `"$x$x"` or `"$x$y$y$x"`:

```
% simple_grep '^(\w+)(\w+)?(?(2)\2\1|\1)$' /usr/dict/words
```

```
beriberi
coco
couscous
deed
...
toot
toto
tutu
```

The lookbehind `condition` allows, along with backreferences, an earlier part of the match to influence a later part of the match. For instance,

```
/[ATGC]+(?(?<=AA)G|C)$/;
```

matches a DNA sequence such that it either ends in `AAG`, or some other base pair combination and `C`. Note that the form is `(?(?<=AA)G|C)` and not `(?((?<=AA))G|C)`; for the lookahead, lookbehind or code assertions, the parentheses around the conditional are not needed.

A bit of magic: executing Perl code in a regular expression

Normally, regexps are a part of Perl expressions. **Code evaluation** expressions turn that around by allowing arbitrary Perl code to be a part of a regexp. A code evaluation expression is denoted `(?{code})`, with `code` a string of Perl statements.

Code expressions are zero-width assertions, and the value they return depends on their environment. There are two possibilities: either the code expression is used as a conditional in a conditional expression `(?(condition)...)`, or it is not. If the code expression is a conditional, the code is evaluated and the result (i.e., the result of the last statement) is used to determine truth or falsehood. If the code expression is not used as a conditional, the assertion always evaluates true and the result is put into the special variable `$^R`. The variable `$^R` can then be used in code expressions later in the regexp. Here are some silly examples:

```
$x = "abcdef";
$x =~ /abc(?{print "Hi Mom!";})def/; # matches,
                                     # prints 'Hi Mom!'
$x =~ /aaa(?{print "Hi Mom!";})def/; # doesn't match,
                                     # no 'Hi Mom!'
```

Pay careful attention to the next example:

```
$x =~ /abc(?{print "Hi Mom!";})ddd/; # doesn't match,
                                     # no 'Hi Mom!'
                                     # but why not?
```

At first glance, you'd think that it shouldn't print, because obviously the `ddd` isn't going to match the target string. But look at this example:

```
$x =~ /abc(?{print "Hi Mom!";})[d]dd/; # doesn't match,
                                       # but _does_ print
```

Hmm. What happened here? If you've been following along, you know that the above pattern should be effectively the same as the last one -- enclosing the d in a character class isn't going to change what it matches. So why does the first not print while the second one does?

The answer lies in the optimizations the REx engine makes. In the first case, all the engine sees are plain old characters (aside from the `?{}` construct). It's smart enough to realize that the string 'ddd' doesn't occur in our target string before actually running the pattern through. But in the second case,

we've tricked it into thinking that our pattern is more complicated than it is. It takes a look, sees our character class, and decides that it will have to actually run the pattern to determine whether or not it matches, and in the process of running it hits the print statement before it discovers that we don't have a match.

To take a closer look at how the engine does optimizations, see the section *Pragmas and debugging* below.

More fun with `?{}`:

```
$x =~ /(?{print "Hi Mom!";})/;       # matches,
                                     # prints 'Hi Mom!'
$x =~ /(?{$c = 1;})(?{print "$c";})/;  # matches,
                                       # prints '1'
$x =~ /(?{$c = 1;})(?{print "$^R";})/; # matches,
                                       # prints '1'
```

The bit of magic mentioned in the section title occurs when the regexp backtracks in the process of searching for a match. If the regexp backtracks over a code expression and if the variables used within are localized using `local`, the changes in the variables produced by the code expression are undone! Thus, if we wanted to count how many times a character got matched inside a group, we could use, e.g.,

```
$x = "aaaa";
$count = 0;  # initialize 'a' count
$c = "bob";  # test if $c gets clobbered
$x =~ /(?{local $c = 0;})         # initialize count
       ( a                        # match 'a'
         (?{local $c = $c + 1;})  # increment count
       )*                         # do this any number of times,
       aa                         # but match 'aa' at the end
       (?{$count = $c;})          # copy local $c var into $count
      /x;
print "'a' count is $count, \$c variable is '$c'\n";
```

This prints

```
'a' count is 2, $c variable is 'bob'
```

If we replace the `(?{local $c = $c + 1;})` with `(?{$c = $c + 1;})`, the variable changes are *not* undone during backtracking, and we get

```
'a' count is 4, $c variable is 'bob'
```

Note that only localized variable changes are undone. Other side effects of code expression execution are permanent. Thus

```
$x = "aaaa";
$x =~ /(a(?{print "Yow\n";}))*aa/;
```

produces

```
Yow
Yow
Yow
Yow
```

The result $^R is automatically localized, so that it will behave properly in the presence of backtracking.

This example uses a code expression in a conditional to match the article 'the' in either English or German:

```
$lang = 'DE';  # use German
...
$text = "das";
print "matched\n"
    if $text =~ /(?(?{
                      $lang eq 'EN'; # is the language English?
                    })
                   the |             # if so, then match 'the'
                   (die|das|der)     # else, match 'die|das|der'
                 )
                /xi;
```

Note that the syntax here is `(?(?{...})yes-regexp|no-regexp)`, not `(?((?{...}))yes-regexp|no-regexp)`. In other words, in the case of a code expression, we don't need the extra parentheses around the conditional.

If you try to use code expressions with interpolating variables, perl may surprise you:

```
$bar = 5;
$pat = '(?{ 1 })';
/foo(?{ $bar })bar/; # compiles ok, $bar not interpolated
/foo(?{ 1 })$bar/;   # compile error!
/foo${pat}bar/;      # compile error!

$pat = qr/(?{ $foo = 1 })/;  # precompile code regexp
/foo${pat}bar/;      # compiles ok
```

If a regexp has (1) code expressions and interpolating variables, or (2) a variable that interpolates a code expression, perl treats the regexp as an error. If the code expression is precompiled into a variable, however, interpolating is ok. The question is, why is this an error?

The reason is that variable interpolation and code expressions together pose a security risk. The combination is dangerous because many programmers who write search engines often take user input and plug it directly into a regexp:

```
$regexp = <>;       # read user-supplied regexp
$chomp $regexp;     # get rid of possible newline
$text =~ /$regexp/; # search $text for the $regexp
```

If the `$regexp` variable contains a code expression, the user could then execute arbitrary Perl code. For instance, some joker could search for `system('rm -rf *');` to erase your files. In this sense, the combination of interpolation and code expressions **taints** your regexp. So by default, using both interpolation and code expressions in the same regexp is not allowed. If you're not concerned about malicious users, it is possible to bypass this security check by invoking `use re 'eval'`:

```
use re 'eval';      # throw caution out the door
$bar = 5;
$pat = '(?{ 1 })';
/foo(?{ 1 })$bar/;  # compiles ok
/foo${pat}bar/;     # compiles ok
```

Another form of code expression is the **pattern code expression** . The pattern code expression is like a regular code expression, except that the result of the code evaluation is treated as a regular expression and matched immediately. A simple example is

```
$length = 5;
$char = 'a';
$x = 'aaaaabb';
$x =~ /(??{$char x $length})/x; # matches, there are 5 of 'a'
```

This final example contains both ordinary and pattern code expressions. It detects if a binary string `1101010010001...` has a Fibonacci spacing 0,1,1,2,3,5,... of the `1`'s:

```
$s0 = 0; $s1 = 1; # initial conditions
$x = "1101010010001000001";
print "It is a Fibonacci sequence\n"
    if $x =~ /^1         # match an initial '1'
                (
                   (??{'0' x $s0}) # match $s0 of '0'
                   1               # and then a '1'
                   (?{
                      $largest = $s0;   # largest seq so far
                      $s2 = $s1 + $s0;  # compute next term
                      $s0 = $s1;        # in Fibonacci sequence
                      $s1 = $s2;
                    })
                  )+   # repeat as needed
               $       # that is all there is
              /x;
print "Largest sequence matched was $largest\n";
```

This prints

```
It is a Fibonacci sequence
Largest sequence matched was 5
```

Ha! Try that with your garden variety regexp package...

Note that the variables `$s0` and `$s1` are not substituted when the regexp is compiled, as happens for ordinary variables outside a code expression. Rather, the code expressions are evaluated when perl encounters them during the search for a match.

The regexp without the `//x` modifier is

```
/^1((??{'0'x$s0})1(?{$largest=$s0;$s2=$s1+$s0$s0=$s1;$s1=$s2;}))+$/;
```

and is a great start on an Obfuscated Perl entry :-) When working with code and conditional expressions, the extended form of regexps is almost necessary in creating and debugging regexps.

Pragmas and debugging

Speaking of debugging, there are several pragmas available to control and debug regexps in Perl. We have already encountered one pragma in the previous section, `use re 'eval';` , that allows variable interpolation and code expressions to coexist in a regexp. The other pragmas are

```
use re 'taint';
$tainted = <>;
@parts = ($tainted =~ /(\w+)\s+(\w+)/; # @parts is now tainted
```

The `taint` pragma causes any substrings from a match with a tainted variable to be tainted as well. This is not normally the case, as regexps are often used to extract the safe bits from a tainted variable. Use `taint` when you are not extracting safe bits, but are performing some other processing. Both `taint` and `eval` pragmas are lexically scoped, which means they are in effect only until the end of the block enclosing the pragmas.

```
use re 'debug';
/^(.*)$/s;       # output debugging info

use re 'debugcolor';
/^(.*)$/s;       # output debugging info in living color
```

The global `debug` and `debugcolor` pragmas allow one to get detailed debugging info about regexp compilation and execution. `debugcolor` is the same as debug, except the debugging information is displayed in color on terminals that can display termcap color sequences. Here is example output:

```
% perl -e 'use re "debug"; "abc" =~ /a*b+c/;'
Compiling REx 'a*b+c'
size 9 first at 1
   1: STAR(4)
   2:   EXACT <a>(0)
   4: PLUS(7)
   5:   EXACT <b>(0)
   7: EXACT <c>(9)
   9: END(0)
floating 'bc' at 0..2147483647 (checking floating) minlen 2
Guessing start of match, REx 'a*b+c' against 'abc'...
Found floating substr 'bc' at offset 1...
Guessed: match at offset 0
Matching REx 'a*b+c' against 'abc'
  Setting an EVAL scope, savestack=3
   0 <> <abc>             |  1:  STAR
                           EXACT <a> can match 1 times out of 32767...
  Setting an EVAL scope, savestack=3
   1 <a> <bc>             |  4:    PLUS
                           EXACT <b> can match 1 times out of 32767...
  Setting an EVAL scope, savestack=3
   2 <ab> <c>             |  7:      EXACT <c>
   3 <abc> <>             |  9:      END
Match successful!
Freeing REx: 'a*b+c'
```

If you have gotten this far into the tutorial, you can probably guess what the different parts of the debugging output tell you. The first part

```
Compiling REx 'a*b+c'
size 9 first at 1
   1: STAR(4)
   2:   EXACT <a>(0)
   4: PLUS(7)
   5:   EXACT <b>(0)
   7: EXACT <c>(9)
   9: END(0)
```

describes the compilation stage. `STAR(4)` means that there is a starred object, in this case `'a'`, and if it matches, goto line 4, i.e., `PLUS(7)`. The middle lines describe some heuristics and optimizations

performed before a match:

```
floating ‘bc’ at 0..2147483647 (checking floating) minlen 2
Guessing start of match, REx ‘a*b+c’ against ‘abc’...
Found floating substr ‘bc’ at offset 1...
Guessed: match at offset 0
```

Then the match is executed and the remaining lines describe the process:

```
Matching REx ‘a*b+c’ against ‘abc’
  Setting an EVAL scope, savestack=3
   0 <> <abc>             |  1:  STAR
                           EXACT <a> can match 1 times out of 32767...
  Setting an EVAL scope, savestack=3
   1 <a> <bc>             |  4:    PLUS
                           EXACT <b> can match 1 times out of 32767...
  Setting an EVAL scope, savestack=3
   2 <ab> <c>             |  7:      EXACT <c>
   3 <abc> <>             |  9:      END
Match successful!
Freeing REx: ‘a*b+c’
```

Each step is of the form n <x> <y> , with <x> the part of the string matched and <y> the part not yet matched. The | 1: STAR says that perl is at line number 1 n the compilation list above. See *"Debugging regular expressions" in perldebguts* for much more detail.

An alternative method of debugging regexps is to embed `print` statements within the regexp. This provides a blow-by-blow account of the backtracking in an alternation:

```
"that this" =~ m@(?{print "Start at position ", pos, "\n";})
                 t(?{print "t1\n";})
                 h(?{print "h1\n";})
                 i(?{print "i1\n";})
                 s(?{print "s1\n";})
                     |
                 t(?{print "t2\n";})
                 h(?{print "h2\n";})
                 a(?{print "a2\n";})
                 t(?{print "t2\n";})
                 (?{print "Done at position ", pos, "\n";})
                @x;
```

prints

```
Start at position 0
t1
h1
t2
h2
a2
t2
Done at position 4
```

BUGS

Code expressions, conditional expressions, and independent expressions are **experimental**. Don't use them in production code. Yet.

SEE ALSO

This is just a tutorial. For the full story on perl regular expressions, see the *perlre* regular expressions reference page.

For more information on the matching `m//` and substitution `s///` operators, see *"Regexp Quote-Like Operators" in perlop*. For information on the `split` operation, see *"split" in perlfunc*.

For an excellent all-around resource on the care and feeding of regular expressions, see the book *Mastering Regular Expressions* by Jeffrey Friedl (published by O'Reilly, ISBN 1556592-257-3).

AUTHOR AND COPYRIGHT

Acknowledgments

The inspiration for the stop codon DNA example came from the ZIP code example in chapter 7 of *Mastering Regular Expressions*.

The author would like to thank Jeff Pinyan, Andrew Johnson, Peter Haworth, Ronald J Kimball, and Joe Smith for all their helpful comments.