NAME

perlre - Perl regular expressions

DESCRIPTION

This page describes the syntax of regular expressions in Perl.

If you haven't used regular expressions before, a quick-start introduction is available in *perlrequick*, and a longer tutorial introduction is available in *perlretut*.

For reference on how regular expressions are used in matching operations, plus various examples of the same, see discussions of `m//`, `s///`, `qr//` and `??` in *"Regexp Quote-Like Operators" in perlop*.

Modifiers

Matching operations can have various modifiers. Modifiers that relate to the interpretation of the regular expression inside are listed below. Modifiers that alter the way a regular expression is used by Perl are detailed in *"Regexp Quote-Like Operators" in perlop* and *"Gory details of parsing quoted constructs" in perlop*.

m

Treat string as multiple lines. That is, change "^" and "$" from matching the start or end of the string to matching the start or end of any line anywhere within the string.

s

Treat string as single line. That is, change "." to match any character whatsoever, even a newline, which normally it would not match.

Used together, as `/ms`, they let the "." match any character whatsoever, while still allowing "^" and "$" to match, respectively, just after and just before newlines within the string.

i

Do case-insensitive pattern matching.

If `use locale` is in effect, the case map is taken from the current locale. See *perllocale*.

x

Extend your pattern's legibility by permitting whitespace and comments.

p

Preserve the string matched such that ${^PREMATCH}, ${^MATCH}, and ${^POSTMATCH} are available for use after matching.

g and c

Global matching, and keep the Current position after failed matching. Unlike i, m, s and x, these two flags affect the way the regex is used rather than the regex itself. See *"Using regular expressions in Perl" in perlretut* for further explanation of the g and c modifiers.

These are usually written as "the `/x` modifier", even though the delimiter in question might not really be a slash. Any of these modifiers may also be embedded within the regular expression itself using the `(?...)` construct. See below.

The `/x` modifier itself needs a little more explanation. It tells the regular expression parser to ignore most whitespace that is neither backslashed nor within a character class. You can use this to break up your regular expression into (slightly) more readable parts. The # character is also treated as a metacharacter introducing a comment, just as in ordinary Perl code. This also means that if you want real whitespace or # characters in the pattern (outside a character class, where they are unaffected by `/x`), then you'll either have to escape them (using backslashes or `\Q...\E`) or encode them using octal, hex, or `\N{}` escapes. Taken together, these features go a long way towards making Perl's regular expressions more readable. Note that you have to be careful not to include the pattern

delimiter in the comment--perl has no way of knowing you did not intend to close the pattern early. See the C-comment deletion code in *perlop*. Also note that anything inside a `\Q...\E` stays unaffected by `/x`. And note that `/x` doesn't affect whether space interpretation within a single multi-character construct. For example in `\x{...}`, regardless of the `/x` modifier, there can be no spaces. Same for a *quantifier* such as `{3}` or `{5,}`. Similarly, `(?:...)` can't have a space between the `?` and `:`, but can between the `(` and `?`. Within any delimiters for such a construct, allowed spaces are not affected by `/x`, and depend on the construct. For example, `\x{...}` can't have spaces because hexadecimal numbers don't have spaces in them. But, Unicode properties can have spaces, so in `\p{...}` there can be spaces that follow the Unicode rules, for which see *"Properties accessible through \p{} and \P{}" in perluniprops.pod*.

Regular Expressions

Metacharacters

The patterns used in Perl pattern matching evolved from those supplied in the Version 8 regex routines. (The routines are derived (distantly) from Henry Spencer's freely redistributable reimplementation of the V8 routines.) See *Version 8 Regular Expressions* for details.

In particular the following metacharacters have their standard *egrep*-ish meanings:

```
\ Quote the next metacharacter
^ Match the beginning of the line
. Match any character (except newline)
$ Match the end of the line (or before newline at the end)
| Alternation
() Grouping
[] Character class
```

By default, the "^" character is guaranteed to match only the beginning of the string, the "$" character only the end (or before the newline at the end), and Perl does certain optimizations with the assumption that the string contains only one line. Embedded newlines will not be matched by "^" or "$". You may, however, wish to treat a string as a multi-line buffer, such that the "^" will match after any newline within the string (except if the newline is the last character in the string), and "$" will match before any newline. At the cost of a little more overhead, you can do this by using the /m modifier on the pattern match operator. (Older programs did this by setting `$*`, but this practice has been removed in perl 5.9.)

To simplify multi-line substitutions, the "." character never matches a newline unless you use the `/s` modifier, which in effect tells Perl to pretend the string is a single line--even if it isn't.

Quantifiers

The following standard quantifiers are recognized:

```
*    Match 0 or more times
+    Match 1 or more times
?    Match 1 or 0 times
{n}    Match exactly n times
{n,}   Match at least n times
{n,m}  Match at least n but not more than m times
```

(If a curly bracket occurs in any other context, it is treated as a regular character. In particular, the lower bound is not optional.) The "*" quantifier is equivalent to `{0,}`, the "+" quantifier to `{1,}`, and the "?" quantifier to `{0,1}`. n and m are limited to non-negative integral values less than a preset limit defined when perl is built. This is usually 32766 on the most common platforms. The actual limit can be seen in the error message generated by code such as this:

```
$_ **= $_ , / {$_} / for 2 .. 42;
```

By default, a quantified subpattern is "greedy", that is, it will match as many times as possible (given a particular starting location) while still allowing the rest of the pattern to match. If you want it to match the minimum number of times possible, follow the quantifier with a "?". Note that the meanings don't change, just the "greediness":

```
*?     Match 0 or more times, not greedily
+?     Match 1 or more times, not greedily
??     Match 0 or 1 time, not greedily
{n}?   Match exactly n times, not greedily
{n,}?  Match at least n times, not greedily
{n,m}? Match at least n but not more than m times, not greedily
```

By default, when a quantified subpattern does not allow the rest of the overall pattern to match, Perl will backtrack. However, this behaviour is sometimes undesirable. Thus Perl provides the "possessive" quantifier form as well.

```
*+     Match 0 or more times and give nothing back
++     Match 1 or more times and give nothing back
?+     Match 0 or 1 time and give nothing back
{n}+   Match exactly n times and give nothing back (redundant)
{n,}+  Match at least n times and give nothing back
{n,m}+ Match at least n but not more than m times and give nothing back
```

For instance,

```
'aaaa' =~ /a++a/
```

will never match, as the a++ will gobble up all the a's in the string and won't leave any for the remaining part of the pattern. This feature can be extremely useful to give perl hints about where it shouldn't backtrack. For instance, the typical "match a double-quoted string" problem can be most efficiently performed when written as:

```
/"(?:[^"\\]++|\\.)*+"/
```

as we know that if the final quote does not match, backtracking will not help. See the independent subexpression `(?>...)` for more details; possessive quantifiers are just syntactic sugar for that construct. For instance the above example could also be written as follows:

```
/"(?>(?:(?>[^"\\]+)|\\.)*)"/
```

Escape sequences

Because patterns are processed as double quoted strings, the following also work:

```
\t  tab                   (HT, TAB)
\n  newline               (LF, NL)
\r  return                (CR)
\f  form feed             (FF)
\a  alarm (bell)          (BEL)
\e  escape (think troff)  (ESC)
\033 octal char            (example: ESC)
\x1B hex char              (example: ESC)
\x{263a} long hex char         (example: Unicode SMILEY)
\cK  control char          (example: VT)
\N{name} named Unicode character
\N{U+263D} Unicode character     (example: FIRST QUARTER MOON)
\l  lowercase next char (think vi)
```

```
\u  uppercase next char (think vi)
\L  lowercase till \E (think vi)
\U  uppercase till \E (think vi)
\E  end case modification (think vi)
\Q  quote (disable) pattern metacharacters till \E
```

If `use locale` is in effect, the case map used by `\l`, `\L`, `\u` and `\U` is taken from the current locale. See *perllocale*. For documentation of `\N{name}`, see *charnames*.

You cannot include a literal `$` or `@` within a `\Q` sequence. An unescaped `$` or `@` interpolates the corresponding variable, while escaping will cause the literal string `\$` to be matched. You'll need to write something like `m/\Quser\E\@\Qhost/`.

Character Classes and other Special Escapes

In addition, Perl defines the following:

```
\w      Match a "word" character (alphanumeric plus "_")
\W      Match a non-"word" character
\s      Match a whitespace character
\S      Match a non-whitespace character
\d      Match a digit character
\D      Match a non-digit character
\pP      Match P, named property.  Use \p{Prop} for longer names.
\PP      Match non-P
\X      Match Unicode "eXtended grapheme cluster"
\C      Match a single C char (octet) even under Unicode.
  NOTE: breaks up characters into their UTF-8 bytes,
  so you may end up with malformed pieces of UTF-8.
  Unsupported in lookbehind.
\1       Backreference to a specific group.
  '1' may actually be any positive integer.
\g1      Backreference to a specific or previous group,
\g{-1}   number may be negative indicating a previous buffer and may
         optionally be wrapped in curly brackets for safer parsing.
\g{name} Named backreference
\k<name> Named backreference
\K       Keep the stuff left of the \K, don't include it in $&
\N       Any character but \n (experimental)
\v       Vertical whitespace
\V       Not vertical whitespace
\h       Horizontal whitespace
\H       Not horizontal whitespace
\R       Linebreak
```

See *"Backslashed sequences" in perlrecharclass* for details on `\w`, `\W`, `\s`, `\S`, `\d`, `\D`, `\p`, `\P`, `\N`, `\v`, `\V`, `\h`, and `\H`. See *"Misc" in perlrebackslash* for details on `\R` and `\X`.

Note that `\N` has two meanings. When of the form `\N{NAME}`, it matches the character whose name is `NAME`; and similarly when of the form `\N{U+`*wide hex char*`}`, it matches the character whose Unicode ordinal is *wide hex char*. Otherwise it matches any character but `\n`.

The POSIX character class syntax

```
[:class:]
```

is also available. Note that the `[` and `]` brackets are *literal*; they must always be used within a character class expression.

```
    # this is correct:
    $string =~ /[[:alpha:]]/;

    # this is not, and will generate a warning:
    $string =~ /[:alpha:]/;
```

The following Posix-style character classes are available:

```
 [[:alpha:]]  Any alphabetical character.
 [[:alnum:]]  Any alphanumerical character.
 [[:ascii:]]  Any character in the ASCII character set.
 [[:blank:]]  A GNU extension, equal to a space or a horizontal tab
 [[:cntrl:]]  Any control character.
 [[:digit:]]  Any decimal digit, equivalent to "\d".
 [[:graph:]]  Any printable character, excluding a space.
 [[:lower:]]  Any lowercase character.
 [[:print:]]  Any printable character, including a space.
 [[:punct:]]  Any graphical character excluding "word" characters.
 [[:space:]]  Any whitespace character. "\s" plus vertical tab ("\cK").
 [[:upper:]]  Any uppercase character.
 [[:word:]]   A Perl extension, equivalent to "\w".
 [[:xdigit:]] Any hexadecimal digit.
```

You can negate the [::] character classes by prefixing the class name with a '^'. This is a Perl extension.

The POSIX character classes [.cc.] and [=cc=] are recognized but **not** supported and trying to use them will cause an error.

Details on POSIX character classes are in *"Posix Character Classes" in perlrecharclass*.

Assertions

Perl defines the following zero-width assertions:

```
    \b Match a word boundary
    \B Match except at a word boundary
    \A Match only at beginning of string
    \Z Match only at end of string, or before newline at the end
    \z Match only at end of string
    \G Match only at pos() (e.g. at the end-of-match position
        of prior m//g)
```

A word boundary (`\b`) is a spot between two characters that has a `\w` on one side of it and a `\W` on the other side of it (in either order), counting the imaginary characters off the beginning and end of the string as matching a `\W`. (Within character classes `\b` represents backspace rather than a word boundary, just as it normally does in any double-quoted string.) The `\A` and `\Z` are just like "^" and "$", except that they won't match multiple times when the `/m` modifier is used, while "^" and "$" will match at every internal line boundary. To match the actual end of the string and not ignore an optional trailing newline, use `\z`.

The `\G` assertion can be used to chain global matches (using `m//g`), as described in *"Regexp Quote-Like Operators" in perlop*. It is also useful when writing `lex`-like scanners, when you have several patterns that you want to match against consequent substrings of your string, see the previous reference. The actual location where `\G` will match can also be influenced by using `pos()` as an lvalue: see *"pos" in perlfunc*. Note that the rule for zero-length matches is modified somewhat, in that contents to the left of `\G` is not counted when determining the length of the match. Thus the

following will not match forever:

```
$str = 'ABC';
pos($str) = 1;
while (/.\G/g) {
    print $&;
}
```

It will print 'A' and then terminate, as it considers the match to be zero-width, and thus will not match at the same position twice in a row.

It is worth noting that `\G` improperly used can result in an infinite loop. Take care when using patterns that include `\G` in an alternation.

Capture buffers

The bracketing construct `( ... )` creates capture buffers. To refer to the current contents of a buffer later on, within the same pattern, use \1 for the first, \2 for the second, and so on. Outside the match use "$" instead of "\". (The \<digit> notation works in certain circumstances outside the match. See the warning below about \1 vs $1 for details.) Referring back to another part of the match is called a *backreference*.

There is no limit to the number of captured substrings that you may use. However Perl also uses \10, \11, etc. as aliases for \010, \011, etc. (Recall that 0 means octal, so \011 is the character at number 9 in your coded character set; which would be the 10th character, a horizontal tab under ASCII.) Perl resolves this ambiguity by interpreting \10 as a backreference only if at least 10 left parentheses have opened before it. Likewise \11 is a backreference only if at least 11 left parentheses have opened before it. And so on. \1 through \9 are always interpreted as backreferences. If the bracketing group did not match, the associated backreference won't match either. (This can happen if the bracketing group is optional, or in a different branch of an alternation.)

In order to provide a safer and easier way to construct patterns using backreferences, Perl provides the `\g{N}` notation (starting with perl 5.10.0). The curly brackets are optional, however omitting them is less safe as the meaning of the pattern can be changed by text (such as digits) following it. When N is a positive integer the `\g{N}` notation is exactly equivalent to using normal backreferences. When N is a negative integer then it is a relative backreference referring to the previous N'th capturing group. When the bracket form is used and N is not an integer, it is treated as a reference to a named buffer.

Thus `\g{-1}` refers to the last buffer, `\g{-2}` refers to the buffer before that. For example:

```
/
 (Y)            # buffer 1
 (              # buffer 2
    (X)         # buffer 3
    \g{-1}      # backref to buffer 3
    \g{-3}      # backref to buffer 1
 )
/x
```

and would match the same as `/(Y) ( (X) \3 \1 )/x`.

Additionally, as of Perl 5.10.0 you may use named capture buffers and named backreferences. The notation is `(?<name>...)` to declare and `\k<name>` to reference. You may also use apostrophes instead of angle brackets to delimit the name; and you may use the bracketed `\g{name}` backreference syntax. It's possible to refer to a named capture buffer by absolute and relative number as well. Outside the pattern, a named capture buffer is available via the `%+` hash. When different buffers within the same pattern have the same name, `$+{name}` and `\k<name>` refer to the leftmost defined group. (Thus it's possible to do things with named capture buffers that would otherwise

require (??{}) code to accomplish.)

Examples:

```
    s/^([^ ]*) *([^ ]*)/$2 $1/;     # swap first two words

    /(.)\1/                         # find first doubled char
         and print "'$1' is the first doubled character\n";

    /(?<char>.)\k<char>/            # ... a different way
         and print "'$+{char}' is the first doubled character\n";

    /(?'char'.)\1/                  # ... mix and match
         and print "'$1' is the first doubled character\n";

    if (/Time: (..):(..):(..)/) {   # parse out values
 $hours = $1;
 $minutes = $2;
 $seconds = $3;
    }
```

Several special variables also refer back to portions of the previous match. $+ returns whatever the last bracket match matched. $& returns the entire matched string. (At one point $0 did also, but now it returns the name of the program.) $` returns everything before the matched string. $' returns everything after the matched string. And $^N contains whatever was matched by the most-recently closed group (submatch). $^N can be used in extended patterns (see below), for example to assign a submatch to a variable.

The numbered match variables ($1, $2, $3, etc.) and the related punctuation set ($+, $&, $`, $', and $^N) are all dynamically scoped until the end of the enclosing block or until the next successful match, whichever comes first. (See *"Compound Statements" in perlsyn*.)

NOTE: Failed matches in Perl do not reset the match variables, which makes it easier to write code that tests for a series of more specific cases and remembers the best match.

WARNING: Once Perl sees that you need one of $&, $`, or $' anywhere in the program, it has to provide them for every pattern match. This may substantially slow your program. Perl uses the same mechanism to produce $1, $2, etc, so you also pay a price for each pattern that contains capturing parentheses. (To avoid this cost while retaining the grouping behaviour, use the extended regular expression (?: ...) instead.) But if you never use $&, $` or $', then patterns *without* capturing parentheses will not be penalized. So avoid $&, $', and $` if you can, but if you can't (and some algorithms really appreciate them), once you've used them once, use them at will, because you've already paid the price. As of 5.005, $& is not so costly as the other two.

As a workaround for this problem, Perl 5.10.0 introduces ${^PREMATCH}, ${^MATCH} and ${^POSTMATCH}, which are equivalent to $`, $& and $', **except** that they are only guaranteed to be defined after a successful match that was executed with the /p (preserve) modifier. The use of these variables incurs no global performance penalty, unlike their punctuation char equivalents, however at the trade-off that you have to tell perl when you want to use them.

Backslashed metacharacters in Perl are alphanumeric, such as \b, \w, \n. Unlike some other regular expression languages, there are no backslashed symbols that aren't alphanumeric. So anything that looks like \\, \(, \), \<, \>, \{, or \} is always interpreted as a literal character, not a metacharacter. This was once used in a common idiom to disable or quote the special meanings of regular expression metacharacters in a string that you want to use for a pattern. Simply quote all non-"word" characters:

```
    $pattern =~ s/(\W)/\\$1/g;
```

(If `use locale` is set, then this depends on the current locale.) Today it is more common to use the quotemeta() function or the `\Q` metaquoting escape sequence to disable all metacharacters' special meanings like this:

```
/$unquoted\Q$quoted\E$unquoted/
```

Beware that if you put literal backslashes (those not inside interpolated variables) between `\Q` and `\E`, double-quotish backslash interpolation may lead to confusing results. If you *need* to use literal backslashes within `\Q...\E`, consult *"Gory details of parsing quoted constructs" in perlop*.

Extended Patterns

Perl also defines a consistent extension syntax for features not found in standard tools like **awk** and **lex**. The syntax is a pair of parentheses with a question mark as the first thing within the parentheses. The character after the question mark indicates the extension.

The stability of these extensions varies widely. Some have been part of the core language for many years. Others are experimental and may change without warning or be completely removed. Check the documentation on an individual feature to verify its current status.

A question mark was chosen for this and for the minimal-matching construct because 1) question marks are rare in older regular expressions, and 2) whenever you see one, you should stop and "question" exactly what is going on. That's psychology...

`(?#text)`

A comment. The text is ignored. If the `/x` modifier enables whitespace formatting, a simple # will suffice. Note that Perl closes the comment as soon as it sees a `)`, so there is no way to put a literal `)` in the comment.

`(?pimsx-imsx)`

One or more embedded pattern-match modifiers, to be turned on (or turned off, if preceded by `-`) for the remainder of the pattern or the remainder of the enclosing pattern group (if any). This is particularly useful for dynamic patterns, such as those read in from a configuration file, taken from an argument, or specified in a table somewhere. Consider the case where some patterns want to be case sensitive and some do not: The case insensitive ones merely need to include `(?i)` at the front of the pattern. For example:

```
$pattern = "foobar";
if ( /$pattern/i ) { }

# more flexible:

$pattern = "(?i)foobar";
if ( /$pattern/ ) { }
```

These modifiers are restored at the end of the enclosing group. For example,

```
( (?i) blah ) \s+ \1
```

will match `blah` in any case, some spaces, and an exact (*including the case*!) repetition of the previous word, assuming the `/x` modifier, and no `/i` modifier outside this group.

These modifiers do not carry over into named subpatterns called in the enclosing group. In other words, a pattern such as `((?i)(&NAME))` does not change the case-sensitivity of the "NAME" pattern.

Note that the p modifier is special in that it can only be enabled, not disabled, and that its presence anywhere in a pattern has a global effect. Thus `(?-p)` and

`(?-p:...)` are meaningless and will warn when executed under `use warnings`.

`(?:pattern)`

`(?imsx-imsx:pattern)`

This is for clustering, not capturing; it groups subexpressions like "()", but doesn't make backreferences as "()" does. So

```
@fields = split(/\b(?:a|b|c)\b/)
```

is like

```
@fields = split(/\b(a|b|c)\b/)
```

but doesn't spit out extra fields. It's also cheaper not to capture characters if you don't need to.

Any letters between ? and : act as flags modifiers as with `(?imsx-imsx)`. For example,

```
/(?s-i:more.*than).*million/i
```

is equivalent to the more verbose

```
/(?:(?s-i)more.*than).*million/i
```

`(?|pattern)`

This is the "branch reset" pattern, which has the special property that the capture buffers are numbered from the same starting point in each alternation branch. It is available starting from perl 5.10.0.

Capture buffers are numbered from left to right, but inside this construct the numbering is restarted for each branch.

The numbering within each branch will be as normal, and any buffers following this construct will be numbered as though the construct contained only one branch, that being the one with the most capture buffers in it.

This construct will be useful when you want to capture one of a number of alternative matches.

Consider the following pattern. The numbers underneath show in which buffer the captured content will be stored.

```
# before  ---------------branch-reset----------- after
/ ( a )  (?| x ( y ) z | (p (q) r) | (t) u (v) ) ( z ) /x
# 1            2         2  3        2     3     4
```

Be careful when using the branch reset pattern in combination with named captures. Named captures are implemented as being aliases to numbered buffers holding the captures, and that interferes with the implementation of the branch reset pattern. If you are using named captures in a branch reset pattern, it's best to use the same names, in the same order, in each of the alternations:

```
/(?|  (?<a> x ) (?<b> y )
   |  (?<a> z ) (?<b> w )) /x
```

Not doing so may lead to surprises:

```
"12" =~ /(?| (?<a> \d+ ) | (?<b> \D+))/x;
say $+ {a};   # Prints '12'
say $+ {b};   # *Also* prints '12'.
```

The problem here is that both the buffer named a and the buffer named b are

aliases for the buffer belonging to $1.

Look-Around Assertions

Look-around assertions are zero width patterns which match a specific pattern without including it in $&. Positive assertions match when their subpattern matches, negative assertions match when their subpattern fails. Look-behind matches text up to the current match position, look-ahead matches text following the current match position.

(?=pattern)

A zero-width positive look-ahead assertion. For example, /\w+(?=\t)/ matches a word followed by a tab, without including the tab in $&.

(?!pattern)

A zero-width negative look-ahead assertion. For example /foo(?!bar)/ matches any occurrence of "foo" that isn't followed by "bar". Note however that look-ahead and look-behind are NOT the same thing. You cannot use this for look-behind.

If you are looking for a "bar" that isn't preceded by a "foo", /(?!foo)bar/ will not do what you want. That's because the (?!foo) is just saying that the next thing cannot be "foo"--and it's not, it's a "bar", so "foobar" will match. You would have to do something like /(?!foo)...bar/ for that. We say "like" because there's the case of your "bar" not having three characters before it. You could cover that this way: /(?:(?!foo)...|^.{0,2})bar/. Sometimes it's still easier just to say:

```
if (/bar/ && $` !~ /foo$/)
```

For look-behind see below.

(?<=pattern) \K

A zero-width positive look-behind assertion. For example, /(?<=\t)\w+/ matches a word that follows a tab, without including the tab in $&. Works only for fixed-width look-behind.

There is a special form of this construct, called \K, which causes the regex engine to "keep" everything it had matched prior to the \K and not include it in $&. This effectively provides variable length look-behind. The use of \K inside of another look-around assertion is allowed, but the behaviour is currently not well defined.

For various reasons \K may be significantly more efficient than the equivalent (?<=...) construct, and it is especially useful in situations where you want to efficiently remove something following something else in a string. For instance

```
s/(foo)bar/$1/g;
```

can be rewritten as the much more efficient

```
s/foo\Kbar//g;
```

(?<!pattern)

A zero-width negative look-behind assertion. For example /(?<!bar)foo/ matches any occurrence of "foo" that does not follow "bar". Works only for fixed-width look-behind.

(?'NAME'pattern)

`(?<NAME>pattern)`

A named capture buffer. Identical in every respect to normal capturing parentheses `()` but for the additional fact that `%+` or `%-` may be used after a successful match to refer to a named buffer. See `perlvar` for more details on the `%+` and `%-` hashes.

If multiple distinct capture buffers have the same name then the $+{NAME} will refer to the leftmost defined buffer in the match.

The forms `(?'NAME'pattern)` and `(?<NAME>pattern)` are equivalent.

NOTE: While the notation of this construct is the same as the similar function in .NET regexes, the behavior is not. In Perl the buffers are numbered sequentially regardless of being named or not. Thus in the pattern

```
/(x)(?<foo>y)(z)/
```

$+{foo} will be the same as $2, and $3 will contain 'z' instead of the opposite which is what a .NET regex hacker might expect.

Currently NAME is restricted to simple identifiers only. In other words, it must match `/^[_A-Za-z][_A-Za-z0-9]*\z/` or its Unicode extension (see *utf8*), though it isn't extended by the locale (see *perllocale*).

NOTE: In order to make things easier for programmers with experience with the Python or PCRE regex engines, the pattern `(?P<NAME>pattern)` may be used instead of `(?<NAME>pattern)`; however this form does not support the use of single quotes as a delimiter for the name.

`\k<NAME>`

`\k'NAME'`

Named backreference. Similar to numeric backreferences, except that the group is designated by name and not number. If multiple groups have the same name then it refers to the leftmost defined group in the current match.

It is an error to refer to a name not defined by a `(?<NAME>)` earlier in the pattern.

Both forms are equivalent.

NOTE: In order to make things easier for programmers with experience with the Python or PCRE regex engines, the pattern `(?P=NAME)` may be used instead of `\k<NAME>`.

`(?{ code })`

WARNING: This extended regular expression feature is considered experimental, and may be changed without notice. Code executed that has side effects may not perform identically from version to version due to the effect of future optimisations in the regex engine.

This zero-width assertion evaluates any embedded Perl code. It always succeeds, and its `code` is not interpolated. Currently, the rules to determine where the `code` ends are somewhat convoluted.

This feature can be used together with the special variable `$^N` to capture the results of submatches in variables without having to keep track of the number of nested parentheses. For example:

```
$_ = "The brown fox jumps over the lazy dog";
/the (\S+)(?{ $color = $^N }) (\S+)(?{ $animal = $^N })/i;
print "color = $color, animal = $animal\n";
```

Inside the `(?{...})` block, `$_` refers to the string the regular expression is matching against. You can also use `pos()` to know what is the current position of matching within this string.

The code is properly scoped in the following sense: If the assertion is backtracked (compare *Backtracking*), all changes introduced after localization are undone, so that

```
  $_ = 'a' x 8;
  m<
     (?{ $cnt = 0 })   # Initialize $cnt.
     (
       a
       (?{
           local $cnt = $cnt + 1; # Update $cnt,
backtracking-safe.
       })
     )*
     aaaa
     (?{ $res = $cnt })   # On success copy to non-localized
     # location.
   >x;
```

will set `$res = 4`. Note that after the match, `$cnt` returns to the globally introduced value, because the scopes that restrict `local` operators are unwound.

This assertion may be used as a `(?(condition)yes-pattern|no-pattern)` switch. If *not* used in this way, the result of evaluation of code is put into the special variable `$^R`. This happens immediately, so `$^R` can be used from other `(?{ code })` assertions inside the same regular expression.

The assignment to `$^R` above is properly localized, so the old value of `$^R` is restored if the assertion is backtracked; compare *Backtracking*.

For reasons of security, this construct is forbidden if the regular expression involves run-time interpolation of variables, unless the perilous `use re 'eval'` pragma has been used (see *re*), or the variables contain results of `qr//` operator (see *"qr/STRING/imosx" in perlop*).

This restriction is due to the wide-spread and remarkably convenient custom of using run-time determined strings as patterns. For example:

```
    $re = <>;
    chomp $re;
    $string =~ /$re/;
```

Before Perl knew how to execute interpolated code within a pattern, this operation was completely safe from a security point of view, although it could raise an exception from an illegal pattern. If you turn on the `use re 'eval'`, though, it is no longer secure, so you should only do so if you are also using taint checking. Better yet, use the carefully constrained evaluation within a Safe compartment. See *perlsec* for details about both these mechanisms.

WARNING: Use of lexical (`my`) variables in these blocks is broken. The result is unpredictable and will make perl unstable. The workaround is to use global (`our`) variables.

WARNING: Because Perl's regex engine is currently not re-entrant, interpolated code may not invoke the regex engine either directly with `m//` or `s///`), or indirectly with functions such as `split`. Invoking the regex engine in these blocks will make perl unstable.

`(??{ code })`

WARNING: This extended regular expression feature is considered experimental, and may be changed without notice. Code executed that has side effects may not

perform identically from version to version due to the effect of future optimisations in the regex engine.

This is a "postponed" regular subexpression. The `code` is evaluated at run time, at the moment this subexpression may match. The result of evaluation is considered as a regular expression and matched as if it were inserted instead of this construct. Note that this means that the contents of capture buffers defined inside an eval'ed pattern are not available outside of the pattern, and vice versa, there is no way for the inner pattern to refer to a capture buffer defined outside. Thus,

```
('a' x 100)=~/(??{'(.)' x 100})/
```

will match, it will **not** set $1.

The `code` is not interpolated. As before, the rules to determine where the `code` ends are currently somewhat convoluted.

The following pattern matches a parenthesized group:

```
$re = qr{
    \(
    (?:
(?> [^()]+ ) # Non-parens without backtracking
     |
(??{ $re }) # Group with matching parens
    )*
    \)
 }x;
```

See also `(?PARNO)` for a different, more efficient way to accomplish the same task.

For reasons of security, this construct is forbidden if the regular expression involves run-time interpolation of variables, unless the perilous `use re 'eval'` pragma has been used (see *re*), or the variables contain results of `qr//` operator (see *"qr/STRING/imosx" in perlop*).

Because perl's regex engine is not currently re-entrant, delayed code may not invoke the regex engine either directly with `m//` or `s///`), or indirectly with functions such as `split`.

Recursing deeper than 50 times without consuming any input string will result in a fatal error. The maximum depth is compiled into perl, so changing it requires a custom build.

`(?PARNO) (?-PARNO) (?+PARNO) (?R) (?0)`

Similar to `(??{ code })` except it does not involve compiling any code, instead it treats the contents of a capture buffer as an independent pattern that must match at the current position. Capture buffers contained by the pattern will have the value as determined by the outermost recursion.

PARNO is a sequence of digits (not starting with 0) whose value reflects the paren-number of the capture buffer to recurse to. `(?R)` recurses to the beginning of the whole pattern. `(?0)` is an alternate syntax for `(?R)`. If PARNO is preceded by a plus or minus sign then it is assumed to be relative, with negative numbers indicating preceding capture buffers and positive ones following. Thus `(?-1)` refers to the most recently declared buffer, and `(?+1)` indicates the next buffer to be declared. Note that the counting for relative recursion differs from that of relative backreferences, in that with recursion unclosed buffers **are** included.

The following pattern matches a function foo() which may contain balanced parentheses as the argument.

```
  $re = qr{ (                    # paren group 1 (full
function)
              foo
              (                  # paren group 2 (parens)
                \(
                  (              # paren group 3 (contents of
 parens)
                  (?:
                   (?> [^()]+ )  # Non-parens without
backtracking
                  |
                   (?2)          # Recurse to start of paren
group 2
                  )*
                  )
                \)
              )
            )
          }x;
```

If the pattern was used as follows

```
'foo(bar(baz)+baz(bop))'=~/$re/
    and print "\$1 = $1\n",
              "\$2 = $2\n",
              "\$3 = $3\n";
```

the output produced should be the following:

```
$1 = foo(bar(baz)+baz(bop))
$2 = (bar(baz)+baz(bop))
$3 = bar(baz)+baz(bop)
```

If there is no corresponding capture buffer defined, then it is a fatal error. Recursing deeper than 50 times without consuming any input string will also result in a fatal error. The maximum depth is compiled into perl, so changing it requires a custom build.

The following shows how using negative indexing can make it easier to embed recursive patterns inside of a `qr//` construct for later use:

```
my $parens = qr/(\((?:[^()]++|(?-1))*+\))/;
if (/foo $parens \s+ + \s+ bar $parens/x) {
   # do something here...
}
```

Note that this pattern does not behave the same way as the equivalent PCRE or Python construct of the same form. In Perl you can backtrack into a recursed group, in PCRE and Python the recursed into group is treated as atomic. Also, modifiers are resolved at compile time, so constructs like (?i:(?1)) or (?:(?i)(?1)) do not affect how the sub-pattern will be processed.

`(?&NAME)`

Recurse to a named subpattern. Identical to `(?PARNO)` except that the parenthesis to recurse to is determined by name. If multiple parentheses have the same name, then it recurses to the leftmost.

It is an error to refer to a name that is not declared somewhere in the pattern.

NOTE: In order to make things easier for programmers with experience with the

Python or PCRE regex engines the pattern `(?P>NAME)` may be used instead of `(?&NAME)`.

`(?(condition)yes-pattern|no-pattern)`

`(?(condition)yes-pattern)`

Conditional expression. `(condition)` should be either an integer in parentheses (which is valid if the corresponding pair of parentheses matched), a look-ahead/look-behind/evaluate zero-width assertion, a name in angle brackets or single quotes (which is valid if a buffer with the given name matched), or the special symbol (R) (true when evaluated inside of recursion or eval). Additionally the R may be followed by a number, (which will be true when evaluated when recursing inside of the appropriate group), or by `&NAME`, in which case it will be true only when evaluated during recursion in the named group.

Here's a summary of the possible predicates:

(1) (2) ...

Checks if the numbered capturing buffer has matched something.

(<NAME>) ('NAME')

Checks if a buffer with the given name has matched something.

(?{ CODE })

Treats the code block as the condition.

(R)

Checks if the expression has been evaluated inside of recursion.

(R1) (R2) ...

Checks if the expression has been evaluated while executing directly inside of the n-th capture group. This check is the regex equivalent of

```
if ((caller(0))[3] eq 'subname') { ... }
```

In other words, it does not check the full recursion stack.

(R&NAME)

Similar to `(R1)`, this predicate checks to see if we're executing directly inside of the leftmost group with a given name (this is the same logic used by `(?&NAME)` to disambiguate). It does not check the full stack, but only the name of the innermost active recursion.

(DEFINE)

In this case, the yes-pattern is never directly executed, and no no-pattern is allowed. Similar in spirit to `(?{0})` but more efficient. See below for details.

For example:

```
m{ ( \( )?
   [^()]+
   (?(1) \) )
 }x
```

matches a chunk of non-parentheses, possibly included in parentheses themselves.

A special form is the `(DEFINE)` predicate, which never executes directly its yes-pattern, and does not allow a no-pattern. This allows to define subpatterns

which will be executed only by using the recursion mechanism. This way, you can define a set of regular expression rules that can be bundled into any pattern you choose.

It is recommended that for this usage you put the DEFINE block at the end of the pattern, and that you name any subpatterns defined within it.

Also, it's worth noting that patterns defined this way probably will not be as efficient, as the optimiser is not very clever about handling them.

An example of how this might be used is as follows:

```
/(?<NAME>(?&NAME_PAT))(?<ADDR>(?&ADDRESS_PAT))
 (?(DEFINE)
   (?<NAME_PAT>....)
   (?<ADRESS_PAT>....)
 )/x
```

Note that capture buffers matched inside of recursion are not accessible after the recursion returns, so the extra layer of capturing buffers is necessary. Thus `$+{NAME_PAT}` would not be defined even though `$+{NAME}` would be.

`(?>pattern)`

An "independent" subexpression, one which matches the substring that a *standalone* `pattern` would match if anchored at the given position, and it matches *nothing other than this substring*. This construct is useful for optimizations of what would otherwise be "eternal" matches, because it will not backtrack (see *Backtracking*). It may also be useful in places where the "grab all you can, and do not give anything back" semantic is desirable.

For example: `^(?>a*)ab` will never match, since `(?>a*)` (anchored at the beginning of string, as above) will match *all* characters a at the beginning of string, leaving no a for `ab` to match. In contrast, `a*ab` will match the same as `a+b`, since the match of the subgroup `a*` is influenced by the following group `ab` (see *Backtracking*). In particular, `a*` inside `a*ab` will match fewer characters than a standalone `a*`, since this makes the tail match.

An effect similar to `(?>pattern)` may be achieved by writing `(?=(pattern))\1` . This matches the same substring as a standalone `a+`, and the following `\1` eats the matched string; it therefore makes a zero-length assertion into an analogue of `(?>...)`. (The difference between these two constructs is that the second one uses a capturing group, thus shifting ordinals of backreferences in the rest of a regular expression.)

Consider this pattern:

```
m{ \(
      (
        [^()]+  # x+
      |
        \( [^()]* \)
      )+
   \)
 }x
```

That will efficiently match a nonempty group with matching parentheses two levels deep or less. However, if there is no such group, it will take virtually forever on a long string. That's because there are so many different ways to split a long string into several substrings. This is what `(.+)+` is doing, and `(.+)+` is similar to a subpattern of the above pattern. Consider how the pattern above detects no-match on `(((()aaaaaaaaaaaaaaaaa` in several seconds, but that each extra letter

doubles this time. This exponential performance will make it appear that your program has hung. However, a tiny change to this pattern

```
m{ \(
      (
        (?> [^()]+ ) # change x+ above to (?> x+ )
      |
        \( [^()]* \)
      )+
   \)
 }x
```

which uses `(?>...)` matches exactly when the one above does (verifying this yourself would be a productive exercise), but finishes in a fourth the time when used on a similar string with 1000000 as. Be aware, however, that this pattern currently triggers a warning message under the `use warnings` pragma or **-w** switch saying it "`matches null string many times in regex`".

On simple groups, such as the pattern `(?> [^()]+ )`, a comparable effect may be achieved by negative look-ahead, as in `[^()]+ (?! [^()] )`. This was only 4 times slower on a string with 1000000 as.

The "grab all you can, and do not give anything back" semantic is desirable in many situations where on the first sight a simple `()*` looks like the correct solution. Suppose we parse text with comments being delimited by # followed by some optional (horizontal) whitespace. Contrary to its appearance, `#[ \t]*` *is not* the correct subexpression to match the comment delimiter, because it may "give up" some whitespace if the remainder of the pattern can be made to match that way. The correct answer is either one of these:

```
(?>#[ \t]*)
#[ \t]*(?![ \t])
```

For example, to grab non-empty comments into $1, one should use either one of these:

```
/ (?> \# [ \t]* ) (         .+ ) /x;
/     \# [ \t]*   ( [^ \t] .* ) /x;
```

Which one you pick depends on which of these expressions better reflects the above specification of comments.

In some literature this construct is called "atomic matching" or "possessive matching".

Possessive quantifiers are equivalent to putting the item they are applied to inside of one of these constructs. The following equivalences apply:

```
Quantifier Form     Bracketing Form
---------------     ---------------
PAT*+               (?>PAT*)
PAT++               (?>PAT+)
PAT?+               (?>PAT?)
PAT{min,max}+       (?>PAT{min,max})
```

Special Backtracking Control Verbs

WARNING: These patterns are experimental and subject to change or removal in a future version of Perl. Their usage in production code should be noted to avoid problems during upgrades.

These special patterns are generally of the form `(*VERB:ARG)`. Unless otherwise stated the ARG argument is optional; in some cases, it is forbidden.

Any pattern containing a special backtracking verb that allows an argument has the special behaviour that when executed it sets the current package's $REGERROR and $REGMARK variables. When doing so the following rules apply:

On failure, the $REGERROR variable will be set to the ARG value of the verb pattern, if the verb was involved in the failure of the match. If the ARG part of the pattern was omitted, then $REGERROR will be set to the name of the last (*MARK:NAME) pattern executed, or to TRUE if there was none. Also, the $REGMARK variable will be set to FALSE.

On a successful match, the $REGERROR variable will be set to FALSE, and the $REGMARK variable will be set to the name of the last (*MARK:NAME) pattern executed. See the explanation for the (*MARK:NAME) verb below for more details.

NOTE: $REGERROR and $REGMARK are not magic variables like $1 and most other regex related variables. They are not local to a scope, nor readonly, but instead are volatile package variables similar to $AUTOLOAD. Use `local` to localize changes to them to a specific scope if necessary.

If a pattern does not contain a special backtracking verb that allows an argument, then $REGERROR and $REGMARK are not touched at all.

Verbs that take an argument

(*PRUNE) (*PRUNE:NAME)

This zero-width pattern prunes the backtracking tree at the current point when backtracked into on failure. Consider the pattern A (*PRUNE) B, where A and B are complex patterns. Until the (*PRUNE) verb is reached, A may backtrack as necessary to match. Once it is reached, matching continues in B, which may also backtrack as necessary; however, should B not match, then no further backtracking will take place, and the pattern will fail outright at the current starting position.

The following example counts all the possible matching strings in a pattern (without actually matching any of them).

```
'aaab' =~ /a+b?(?{print "$&\n"; $count++})(*FAIL)/;
print "Count=$count\n";
```

which produces:

```
aaab
aaa
aa
a
aab
aa
a
ab
a
Count=9
```

If we add a (*PRUNE) before the count like the following

```
'aaab' =~ /a+b?(*PRUNE)(?{print "$&\n"; $count++})(*FAIL)/;
print "Count=$count\n";
```

we prevent backtracking and find the count of the longest matching at each matching starting point like so:

```
aaab
aab
ab
Count=3
```

Any number of (*PRUNE) assertions may be used in a pattern.

See also (?>pattern) and possessive quantifiers for other ways to control backtracking. In some cases, the use of (*PRUNE) can be replaced with a (?>pattern) with no functional difference; however, (*PRUNE) can be used to handle cases that cannot be expressed using a (?>pattern) alone.

(*SKIP) (*SKIP:NAME)

This zero-width pattern is similar to (*PRUNE), except that on failure it also signifies that whatever text that was matched leading up to the (*SKIP) pattern being executed cannot be part of *any* match of this pattern. This effectively means that the regex engine "skips" forward to this position on failure and tries to match again, (assuming that there is sufficient room to match).

The name of the (*SKIP:NAME) pattern has special significance. If a (*MARK:NAME) was encountered while matching, then it is that position which is used as the "skip point". If no (*MARK) of that name was encountered, then the (*SKIP) operator has no effect. When used without a name the "skip point" is where the match point was when executing the (*SKIP) pattern.

Compare the following to the examples in (*PRUNE), note the string is twice as long:

```
    'aaabaaab' =~ /a+b?(*SKIP)(?{print "$&\n";
$count++})(*FAIL)/;
    print "Count=$count\n";
```

outputs

```
    aaab
    aaab
    Count=2
```

Once the 'aaab' at the start of the string has matched, and the (*SKIP) executed, the next starting point will be where the cursor was when the (*SKIP) was executed.

(*MARK:NAME) (*:NAME) (*MARK:NAME) (*:NAME)

This zero-width pattern can be used to mark the point reached in a string when a certain part of the pattern has been successfully matched. This mark may be given a name. A later (*SKIP) pattern will then skip forward to that point if backtracked into on failure. Any number of (*MARK) patterns are allowed, and the NAME portion may be duplicated.

In addition to interacting with the (*SKIP) pattern, (*MARK:NAME) can be used to "label" a pattern branch, so that after matching, the program can determine which branches of the pattern were involved in the match.

When a match is successful, the $REGMARK variable will be set to the name of the most recently executed (*MARK:NAME) that was involved in the match.

This can be used to determine which branch of a pattern was matched without using a separate capture buffer for each branch, which in turn can result in a performance improvement, as perl cannot optimize /(?:(x)|(y)|(z))/ as efficiently as something like /(?:x(*MARK:x)|y(*MARK:y)|z(*MARK:z))/.

When a match has failed, and unless another verb has been involved in failing the match and has provided its own name to use, the $REGERROR variable will be set to the name of the most recently executed (*MARK:NAME).

See (*SKIP) for more details.

As a shortcut (*MARK:NAME) can be written (*:NAME).

(*THEN) (*THEN:NAME)

This is similar to the "cut group" operator :: from Perl 6. Like (*PRUNE), this verb always matches, and when backtracked into on failure, it causes the regex engine to try the next alternation in the innermost enclosing group (capturing or otherwise).

Its name comes from the observation that this operation combined with the alternation operator (|) can be used to create what is essentially a pattern-based if/then/else block:

```
( COND (*THEN) FOO | COND2 (*THEN) BAR | COND3 (*THEN) BAZ )
```

Note that if this operator is used and NOT inside of an alternation then it acts exactly like the (*PRUNE) operator.

```
/ A (*PRUNE) B /
```

is the same as

```
/ A (*THEN) B /
```

but

```
/ ( A (*THEN) B | C (*THEN) D ) /
```

is not the same as

```
/ ( A (*PRUNE) B | C (*PRUNE) D ) /
```

as after matching the A but failing on the B the (*THEN) verb will backtrack and try C; but the (*PRUNE) verb will simply fail.

(*COMMIT)

This is the Perl 6 "commit pattern" <commit> or :::. It's a zero-width pattern similar to (*SKIP), except that when backtracked into on failure it causes the match to fail outright. No further attempts to find a valid match by advancing the start pointer will occur again. For example,

```
    'aaabaaab' =~ /a+b?(*COMMIT)(?{print "$&\n";
$count++})(*FAIL)/;
    print "Count=$count\n";
```

outputs

```
    aaab
    Count=1
```

In other words, once the (*COMMIT) has been entered, and if the pattern does not match, the regex engine will not try any further matching on the rest of the string.

Verbs without an argument

(*FAIL) (*F)

This pattern matches nothing and always fails. It can be used to force the engine to backtrack. It is equivalent to (?!), but easier to read. In fact, (?!) gets optimised into (*FAIL) internally.

It is probably useful only when combined with (?{}) or (??{}).

(*ACCEPT)

WARNING: This feature is highly experimental. It is not recommended for production code.

This pattern matches nothing and causes the end of successful matching at the point at which the (*ACCEPT) pattern was encountered, regardless of whether there is

actually more to match in the string. When inside of a nested pattern, such as recursion, or in a subpattern dynamically generated via `(??{})`, only the innermost pattern is ended immediately.

If the `(*ACCEPT)` is inside of capturing buffers then the buffers are marked as ended at the point at which the `(*ACCEPT)` was encountered. For instance:

```
'AB' =~ /(A (A|B(*ACCEPT)|C) D)(E)/x;
```

will match, and `$1` will be `AB` and `$2` will be `B`, `$3` will not be set. If another branch in the inner parentheses were matched, such as in the string 'ACDE', then the `D` and `E` would have to be matched as well.

Backtracking

NOTE: This section presents an abstract approximation of regular expression behavior. For a more rigorous (and complicated) view of the rules involved in selecting a match among possible alternatives, see *Combining RE Pieces*.

A fundamental feature of regular expression matching involves the notion called *backtracking*, which is currently used (when needed) by all regular non-possessive expression quantifiers, namely `*`, `*?`, `+`, `+?`, `{n,m}`, and `{n,m}?`. Backtracking is often optimized internally, but the general principle outlined here is valid.

For a regular expression to match, the *entire* regular expression must match, not just part of it. So if the beginning of a pattern containing a quantifier succeeds in a way that causes later parts in the pattern to fail, the matching engine backs up and recalculates the beginning part--that's why it's called backtracking.

Here is an example of backtracking: Let's say you want to find the word following "foo" in the string "Food is on the foo table.":

```
    $_ = "Food is on the foo table.";
    if ( /\b(foo)\s+(\w+)/i ) {
 print "$2 follows $1.\n";
    }
```

When the match runs, the first part of the regular expression (`\b(foo)`) finds a possible match right at the beginning of the string, and loads up $1 with "Foo". However, as soon as the matching engine sees that there's no whitespace following the "Foo" that it had saved in $1, it realizes its mistake and starts over again one character after where it had the tentative match. This time it goes all the way until the next occurrence of "foo". The complete regular expression matches this time, and you get the expected output of "table follows foo."

Sometimes minimal matching can help a lot. Imagine you'd like to match everything between "foo" and "bar". Initially, you write something like this:

```
    $_ =  "The food is under the bar in the barn.";
    if ( /foo(.*)bar/ ) {
 print "got <$1>\n";
    }
```

Which perhaps unexpectedly yields:

```
  got <d is under the bar in the >
```

That's because `.*` was greedy, so you get everything between the *first* "foo" and the *last* "bar". Here it's more effective to use minimal matching to make sure you get the text between a "foo" and the first "bar" thereafter.

```
    if ( /foo(.*?)bar/ ) { print "got <$1>\n" }
  got <d is under the >
```

Here's another example. Let's say you'd like to match a number at the end of a string, and you also want to keep the preceding part of the match. So you write this:

```
    $_ = "I have 2 numbers: 53147";
    if ( /(.*)(\d*)/ ) {    # Wrong!
 print "Beginning is <$1>, number is <$2>.\n";
    }
```

That won't work at all, because `.*` was greedy and gobbled up the whole string. As `\d*` can match on an empty string the complete regular expression matched successfully.

```
    Beginning is <I have 2 numbers: 53147>, number is <>.
```

Here are some variants, most of which don't work:

```
    $_ = "I have 2 numbers: 53147";
    @pats = qw{
 (.*)(\d*)
 (.*)(\d+)
 (.*?)(\d*)
 (.*?)(\d+)
 (.*)(\d+)$
 (.*?)(\d+)$
 (.*)\b(\d+)$
 (.*\D)(\d+)$
    };

    for $pat (@pats) {
 printf "%-12s ", $pat;
 if ( /$pat/ ) {
     print "<$1> <$2>\n";
 } else {
     print "FAIL\n";
 }
    }
```

That will print out:

```
    (.*)(\d*)    <I have 2 numbers: 53147> <>
    (.*)(\d+)    <I have 2 numbers: 5314> <7>
    (.*?)(\d*)   <> <>
    (.*?)(\d+)   <I have > <2>
    (.*)(\d+)$   <I have 2 numbers: 5314> <7>
    (.*?)(\d+)$  <I have 2 numbers: > <53147>
    (.*)\b(\d+)$ <I have 2 numbers: > <53147>
    (.*\D)(\d+)$ <I have 2 numbers: > <53147>
```

As you see, this can be a bit tricky. It's important to realize that a regular expression is merely a set of assertions that gives a definition of success. There may be 0, 1, or several different ways that the definition might succeed against a particular string. And if there are multiple ways it might succeed, you need to understand backtracking to know which variety of success you will achieve.

When using look-ahead assertions and negations, this can all get even trickier. Imagine you'd like to

find a sequence of non-digits not followed by "123". You might try to write that as

```
    $_ = "ABC123";
    if ( /^\D*(?!123)/ ) {  # Wrong!
 print "Yup, no 123 in $_\n";
    }
```

But that isn't going to match; at least, not the way you're hoping. It claims that there is no 123 in the string. Here's a clearer picture of why that pattern matches, contrary to popular expectations:

```
    $x = 'ABC123';
    $y = 'ABC445';

    print "1: got $1\n" if $x =~ /^(ABC)(?!123)/;
    print "2: got $1\n" if $y =~ /^(ABC)(?!123)/;

    print "3: got $1\n" if $x =~ /^(\D*)(?!123)/;
    print "4: got $1\n" if $y =~ /^(\D*)(?!123)/;
```

This prints

```
    2: got ABC
    3: got AB
    4: got ABC
```

You might have expected test 3 to fail because it seems to a more general purpose version of test 1. The important difference between them is that test 3 contains a quantifier (`\D*`) and so can use backtracking, whereas test 1 will not. What's happening is that you've asked "Is it true that at the start of $x, following 0 or more non-digits, you have something that's not 123?" If the pattern matcher had let `\D*` expand to "ABC", this would have caused the whole pattern to fail.

The search engine will initially match `\D*` with "ABC". Then it will try to match `(?!123` with "123", which fails. But because a quantifier (`\D*`) has been used in the regular expression, the search engine can backtrack and retry the match differently in the hope of matching the complete regular expression.

The pattern really, *really* wants to succeed, so it uses the standard pattern back-off-and-retry and lets `\D*` expand to just "AB" this time. Now there's indeed something following "AB" that is not "123". It's "C123", which suffices.

We can deal with this by using both an assertion and a negation. We'll say that the first part in $1 must be followed both by a digit and by something that's not "123". Remember that the look-aheads are zero-width expressions--they only look, but don't consume any of the string in their match. So rewriting this way produces what you'd expect; that is, case 5 will fail, but case 6 succeeds:

```
    print "5: got $1\n" if $x =~ /^(\D*)(?=\d)(?!123)/;
    print "6: got $1\n" if $y =~ /^(\D*)(?=\d)(?!123)/;

    6: got ABC
```

In other words, the two zero-width assertions next to each other work as though they're ANDed together, just as you'd use any built-in assertions: `/^$/` matches only if you're at the beginning of the line AND the end of the line simultaneously. The deeper underlying truth is that juxtaposition in regular expressions always means AND, except when you write an explicit OR using the vertical bar. `/ab/` means match "a" AND (then) match "b", although the attempted matches are made at different positions because "a" is not a zero-width assertion, but a one-width assertion.

WARNING: Particularly complicated regular expressions can take exponential time to solve because of the immense number of possible ways they can use backtracking to try for a match. For example, without internal optimizations done by the regular expression engine, this will take a painfully long time to run:

```
'aaaaaaaaaaaa' =~ /((a{0,5}){0,5})*[c]/
```

And if you used *'s in the internal groups instead of limiting them to 0 through 5 matches, then it would take forever--or until you ran out of stack space. Moreover, these internal optimizations are not always applicable. For example, if you put `{0,5}` instead of `*` on the external group, no current optimization is applicable, and the match takes a long time to finish.

A powerful tool for optimizing such beasts is what is known as an "independent group", which does not backtrack (see *(?>pattern)*). Note also that zero-length look-ahead/look-behind assertions will not backtrack to make the tail match, since they are in "logical" context: only whether they match is considered relevant. For an example where side-effects of look-ahead *might* have influenced the following match, see *(?>pattern)*.

Version 8 Regular Expressions

In case you're not familiar with the "regular" Version 8 regex routines, here are the pattern-matching rules not described above.

Any single character matches itself, unless it is a *metacharacter* with a special meaning described here or above. You can cause characters that normally function as metacharacters to be interpreted literally by prefixing them with a "\" (e.g., "\." matches a ".", not any character; "\\" matches a "\"). This escape mechanism is also required for the character used as the pattern delimiter.

A series of characters matches that series of characters in the target string, so the pattern `blurfl` would match "blurfl" in the target string.

You can specify a character class, by enclosing a list of characters in `[]`, which will match any character from the list. If the first character after the "[" is "^", the class matches any character not in the list. Within a list, the "-" character specifies a range, so that `a-z` represents all characters between "a" and "z", inclusive. If you want either "-" or "]" itself to be a member of a class, put it at the start of the list (possibly after a "^"), or escape it with a backslash. "-" is also taken literally when it is at the end of the list, just before the closing "]". (The following all specify the same class of three characters: `[-az]`, `[az-]`, and `[a\-z]`. All are different from `[a-z]`, which specifies a class containing twenty-six characters, even on EBCDIC-based character sets.) Also, if you try to use the character classes `\w`, `\W`, `\s`, `\S`, `\d`, or `\D` as endpoints of a range, the "-" is understood literally.

Note also that the whole range idea is rather unportable between character sets--and even within character sets they may cause results you probably didn't expect. A sound principle is to use only ranges that begin from and end at either alphabetics of equal case ([a-e], [A-E]), or digits ([0-9]). Anything else is unsafe. If in doubt, spell out the character sets in full.

Characters may be specified using a metacharacter syntax much like that used in C: "\n" matches a newline, "\t" a tab, "\r" a carriage return, "\f" a form feed, etc. More generally, *nnn*, where *nnn* is a string of octal digits, matches the character whose coded character set value is *nnn*. Similarly, \x*nn*, where *nn* are hexadecimal digits, matches the character whose numeric value is *nn*. The expression \c*x* matches the character control-*x*. Finally, the "." metacharacter matches any character except "\n" (unless you use `/s`).

You can specify a series of alternatives for a pattern using "|" to separate them, so that `fee|fie|foe` will match any of "fee", "fie", or "foe" in the target string (as would `f(e|i|o)e`). The first alternative includes everything from the last pattern delimiter ("(", "[", or the beginning of the pattern) up to the first "|", and the last alternative contains everything from the last "|" to the next pattern delimiter. That's why it's common practice to include alternatives in parentheses: to minimize confusion about where they start and end.

Alternatives are tried from left to right, so the first alternative found for which the entire expression matches, is the one that is chosen. This means that alternatives are not necessarily greedy. For example: when matching `foo|foot` against "barefoot", only the "foo" part will match, as that is the first alternative tried, and it successfully matches the target string. (This might not seem important, but it is important when you are capturing matched text using parentheses.)

Also remember that "|" is interpreted as a literal within square brackets, so if you write `[fee|fie|foe]` you're really only matching `[feio|]`.

Within a pattern, you may designate subpatterns for later reference by enclosing them in parentheses, and you may refer back to the *n*th subpattern later in the pattern using the metacharacter *n*. Subpatterns are numbered based on the left to right order of their opening parenthesis. A backreference matches whatever actually matched the subpattern in the string being examined, not the rules for that subpattern. Therefore, `(0|0x)\d*\s\1\d*` will match "0x1234 0x4321", but not "0x1234 01234", because subpattern 1 matched "0x", even though the rule `0|0x` could potentially match the leading 0 in the second number.

Warning on \1 Instead of $1

Some people get too used to writing things like:

```
$pattern =~ s/(\W)/\\\1/g;
```

This is grandfathered for the RHS of a substitute to avoid shocking the **sed** addicts, but it's a dirty habit to get into. That's because in PerlThink, the righthand side of an `s///` is a double-quoted string. `\1` in the usual double-quoted string means a control-A. The customary Unix meaning of `\1` is kludged in for `s///`. However, if you get into the habit of doing that, you get yourself into trouble if you then add an `/e` modifier.

```
s/(\d+)/ \1 + 1 /eg;     # causes warning under -w
```

Or if you try to do

```
s/(\d+)/\1000/;
```

You can't disambiguate that by saying `\{1}000`, whereas you can fix it with `${1}000`. The operation of interpolation should not be confused with the operation of matching a backreference. Certainly they mean two different things on the *left* side of the `s///`.

Repeated Patterns Matching a Zero-length Substring

WARNING: Difficult material (and prose) ahead. This section needs a rewrite.

Regular expressions provide a terse and powerful programming language. As with most other power tools, power comes together with the ability to wreak havoc.

A common abuse of this power stems from the ability to make infinite loops using regular expressions, with something as innocuous as:

```
'foo' =~ m{ ( o? )* }x;
```

The o? matches at the beginning of `'foo'`, and since the position in the string is not moved by the match, o? would match again and again because of the * quantifier. Another common way to create a similar cycle is with the looping modifier `//g`:

```
@matches = ( 'foo' =~ m{ o? }xg );
```

or

```
print "match: <$&>\n" while 'foo' =~ m{ o? }xg;
```

or the loop implied by split().

However, long experience has shown that many programming tasks may be significantly simplified by using repeated subexpressions that may match zero-length substrings. Here's a simple example being:

```
@chars = split //, $string;    # // is not magic in split
($whitewashed = $string) =~ s/()/ /g; # parens avoid magic s// /
```

Thus Perl allows such constructs, by *forcefully breaking the infinite loop*. The rules for this are different for lower-level loops given by the greedy quantifiers `*+{}`, and for higher-level ones like the `/g` modifier or split() operator.

The lower-level loops are *interrupted* (that is, the loop is broken) when Perl detects that a repeated expression matched a zero-length substring. Thus

```
m{ (?: NON_ZERO_LENGTH | ZERO_LENGTH )* }x;
```

is made equivalent to

```
m{   (?: NON_ZERO_LENGTH )*
   |
     (?: ZERO_LENGTH )?
 }x;
```

The higher level-loops preserve an additional state between iterations: whether the last match was zero-length. To break the loop, the following match after a zero-length match is prohibited to have a length of zero. This prohibition interacts with backtracking (see *Backtracking*), and so the *second best* match is chosen if the *best* match is of zero length.

For example:

```
$_ = 'bar';
s/\w??/<$&>/g;
```

results in `<><b><><a><><r><>`. At each position of the string the best match given by non-greedy `??` is the zero-length match, and the *second best* match is what is matched by `\w`. Thus zero-length matches alternate with one-character-long matches.

Similarly, for repeated `m/()/g` the second-best match is the match at the position one notch further in the string.

The additional state of being *matched with zero-length* is associated with the matched string, and is reset by each assignment to pos(). Zero-length matches at the end of the previous match are ignored during `split`.

Combining RE Pieces

Each of the elementary pieces of regular expressions which were described before (such as `ab` or `\Z`) could match at most one substring at the given position of the input string. However, in a typical regular expression these elementary pieces are combined into more complicated patterns using combining operators `ST`, `S|T`, `S*` etc (in these examples `S` and `T` are regular subexpressions).

Such combinations can include alternatives, leading to a problem of choice: if we match a regular expression `a|ab` against `"abc"`, will it match substring `"a"` or `"ab"`? One way to describe which substring is actually matched is the concept of backtracking (see *Backtracking*). However, this description is too low-level and makes you think in terms of a particular implementation.

Another description starts with notions of "better"/"worse". All the substrings which may be matched

by the given regular expression can be sorted from the "best" match to the "worst" match, and it is the "best" match which is chosen. This substitutes the question of "what is chosen?" by the question of "which matches are better, and which are worse?".

Again, for elementary pieces there is no such question, since at most one match at a given position is possible. This section describes the notion of better/worse for combining operators. In the description below S and T are regular subexpressions.

`ST`

Consider two possible matches, AB and `A'B'`, A and `A'` are substrings which can be matched by S, B and `B'` are substrings which can be matched by T.

If A is better match for S than `A'`, AB is a better match than `A'B'`.

If A and `A'` coincide: AB is a better match than `AB'` if B is better match for T than `B'`.

`S|T`

When S can match, it is a better match than when only T can match.

Ordering of two matches for S is the same as for S. Similar for two matches for T.

`S{REPEAT_COUNT}`

Matches as `SSS...S` (repeated as many times as necessary).

`S{min,max}`

Matches as `S{max}|S{max-1}|...|S{min+1}|S{min}`.

`S{min,max}?`

Matches as `S{min}|S{min+1}|...|S{max-1}|S{max}`.

`S?`, `S*`, `S+`

Same as `S{0,1}`, `S{0,BIG_NUMBER}`, `S{1,BIG_NUMBER}` respectively.

`S??`, `S*?`, `S+?`

Same as `S{0,1}?`, `S{0,BIG_NUMBER}?`, `S{1,BIG_NUMBER}?` respectively.

`(?>S)`

Matches the best match for S and only that.

`(?=S)`, `(?<=S)`

Only the best match for S is considered. (This is important only if S has capturing parentheses, and backreferences are used somewhere else in the whole regular expression.)

`(?!S)`, `(?<!S)`

For this grouping operator there is no need to describe the ordering, since only whether or not S can match is important.

`(??{ EXPR })`, `(?PARNO)`

The ordering is the same as for the regular expression which is the result of EXPR, or the pattern contained by capture buffer PARNO.

`(?(condition)yes-pattern|no-pattern)`

Recall that which of `yes-pattern` or `no-pattern` actually matches is already determined. The ordering of the matches is the same as for the chosen subexpression.

The above recipes describe the ordering of matches *at a given position*. One more rule is needed to understand how a match is determined for the whole regular expression: a match at an earlier position is always better than a match at a later position.

Creating Custom RE Engines

Overloaded constants (see *overload*) provide a simple way to extend the functionality of the RE engine.

Suppose that we want to enable a new RE escape-sequence `\Y|` which matches at a boundary between whitespace characters and non-whitespace characters. Note that `(?=\S)(?<!\S)|(?!\S)(?<=\S)` matches exactly at these positions, so we want to have each `\Y|` in the place of the more complicated version. We can create a module `customre` to do this:

```
package customre;
use overload;

sub import {
  shift;
  die "No argument to customre::import allowed" if @_;
  overload::constant 'qr' => \&convert;
}

sub invalid { die "/$_[0]/: invalid escape '\\$_[1]'"}

# We must also take care of not escaping the legitimate \\Y|
# sequence, hence the presence of '\\' in the conversion rules.
my %rules = ( '\\' => '\\\\',
'Y|' => qr/(?=\S)(?<!\S)|(?!\S)(?<=\S)/ );
sub convert {
  my $re = shift;
  $re =~ s{
            \\ ( \\ | Y . )
          }
          { $rules{$1} or invalid($re,$1) }sgex;
  return $re;
}
```

Now `use customre` enables the new escape in constant regular expressions, i.e., those without any runtime variable interpolations. As documented in *overload*, this conversion will work only over literal parts of regular expressions. For `\Y|$re\Y|` the variable part of this regular expression needs to be converted explicitly (but only if the special meaning of `\Y|` should be enabled inside $re):

```
use customre;
$re = <>;
chomp $re;
$re = customre::convert $re;
/\Y|$re\Y|/;
```

PCRE/Python Support

As of Perl 5.10.0, Perl supports several Python/PCRE specific extensions to the regex syntax. While Perl programmers are encouraged to use the Perl specific syntax, the following are also accepted:

`(?P<NAME>pattern)`

Define a named capture buffer. Equivalent to `(?<NAME>pattern)`.

`(?P=NAME)`

Backreference to a named capture buffer. Equivalent to `\g{NAME}`.

`(?P>NAME)`

Subroutine call to a named capture buffer. Equivalent to `(?&NAME)`.

BUGS

There are numerous problems with case insensitive matching of characters outside the ASCII range, especially with those whose folds are multiple characters, such as ligatures like `LATIN SMALL LIGATURE FF`.

In a bracketed character class with case insensitive matching, ranges only work for ASCII characters. For example, `m/[\N{CYRILLIC CAPITAL LETTER A}-\N{CYRILLIC CAPITAL LETTER YA}]/i` doesn't match all the Russian upper and lower case letters.

Many regular expression constructs don't work on EBCDIC platforms.

This document varies from difficult to understand to completely and utterly opaque. The wandering prose riddled with jargon is hard to fathom in several places.

This document needs a rewrite that separates the tutorial content from the reference content.

SEE ALSO

perlrequick.

perlretut.

"Regexp Quote-Like Operators" in perlop.

"Gory details of parsing quoted constructs" in perlop.

perlfaq6.

"pos" in perlfunc.

perllocale.

perlebcdic.

Mastering Regular Expressions by Jeffrey Friedl, published by O'Reilly and Associates.