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 line only at the left and right ends of the string to matching them 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 locale matching rules are in effect, the case map is taken from the current locale for code points less than 255, and from Unicode rules for larger code points. However, matches that would cross the Unicode rules/non-Unicode rules boundary (ords 255/256) will not succeed. See *perllocale*.

There are a number of Unicode characters that match multiple characters under `/i`. For example, `LATIN SMALL LIGATURE FI` should match the sequence `fi`. Perl is not currently able to do this when the multiple characters are in the pattern and are split between groupings, or when one or more are quantified. Thus

```
 "\N{LATIN SMALL LIGATURE FI}" =~ /fi/i;          # Matches
 "\N{LATIN SMALL LIGATURE FI}" =~ /[fi][fi]/i;    # Doesn't match!
 "\N{LATIN SMALL LIGATURE FI}" =~ /fi*/i;         # Doesn't match!

 # The below doesn't match, and it isn't clear what $1 and $2 would
 # be even if it did!!
 "\N{LATIN SMALL LIGATURE FI}" =~ /(f)(i)/i;      # Doesn't match!
```

Perl doesn't match multiple characters in a bracketed character class unless the character that maps to them is explicitly mentioned, and it doesn't match them at all if the character class is inverted, which otherwise could be highly confusing. See *"Bracketed Character Classes" in perlrecharclass*, and *"Negation" in perlrecharclass*.

x

Extend your pattern's legibility by permitting whitespace and comments. Details in */x*

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.

a, d, l and u

These modifiers, all new in 5.14, affect which character-set semantics (Unicode, etc.) are used, as described below in *Character set modifiers*.

Regular expression modifiers are usually written in documentation as e.g., "the `/x` modifier", even though the delimiter in question might not really be a slash. The modifiers `/imsxadlup` may also be embedded within the regular expression itself using the `(?...)` construct, see *Extended Patterns* below.

/x

`/x` 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 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 `:`. 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*.

Character set modifiers

`/d`, `/u`, `/a`, and `/l`, available starting in 5.14, are called the character set modifiers; they affect the character set semantics used for the regular expression.

The `/d`, `/u`, and `/l` modifiers are not likely to be of much use to you, and so you need not worry about them very much. They exist for Perl's internal use, so that complex regular expression data structures can be automatically serialized and later exactly reconstituted, including all their nuances. But, since Perl can't keep a secret, and there may be rare instances where they are useful, they are documented here.

The `/a` modifier, on the other hand, may be useful. Its purpose is to allow code that is to work mostly on ASCII data to not have to concern itself with Unicode.

Briefly, `/l` sets the character set to that of whatever **L**ocale is in effect at the time of the execution of the pattern match.

`/u` sets the character set to **U**nicode.

`/a` also sets the character set to Unicode, BUT adds several restrictions for **A**SCII-safe matching.

`/d` is the old, problematic, pre-5.14 **D**efault character set behavior. Its only use is to force that old behavior.

At any given time, exactly one of these modifiers is in effect. Their existence allows Perl to keep the originally compiled behavior of a regular expression, regardless of what rules are in effect when it is actually executed. And if it is interpolated into a larger regex, the original's rules continue to apply to it, and only it.

The `/l` and `/u` modifiers are automatically selected for regular expressions compiled within the scope of various pragmas, and we recommend that in general, you use those pragmas instead of specifying these modifiers explicitly. For one thing, the modifiers affect only pattern matching, and do not extend to even any replacement done, whereas using the pragmas give consistent results for all appropriate operations within their scopes. For example,

```
s/foo/\Ubar/il
```

will match "foo" using the locale's rules for case-insensitive matching, but the `/l` does not affect how the `\U` operates. Most likely you want both of them to use locale rules. To do this, instead compile the regular expression within the scope of `use locale`. This both implicitly adds the `/l` and applies locale rules to the `\U`. The lesson is to `use locale` and not `/l` explicitly.

Similarly, it would be better to use `use feature 'unicode_strings'` instead of,

```
s/foo/\Lbar/iu
```

to get Unicode rules, as the `\L` in the former (but not necessarily the latter) would also use Unicode rules.

More detail on each of the modifiers follows. Most likely you don't need to know this detail for `/l`, `/u`, and `/d`, and can skip ahead to */a*.

/l

means to use the current locale's rules (see *perllocale*) when pattern matching. For example, `\w` will match the "word" characters of that locale, and `"/i"` case-insensitive matching will match according to the locale's case folding rules. The locale used will be the one in effect at the time of execution of the pattern match. This may not be the same as the compilation-time locale, and can differ from one match to another if there is an intervening call of the *setlocale() function*.

Perl only supports single-byte locales. This means that code points above 255 are treated as Unicode no matter what locale is in effect. Under Unicode rules, there are a few case-insensitive matches that cross the 255/256 boundary. These are disallowed under `/l`. For example, 0xFF (on ASCII platforms) does not caselessly match the character at 0x178, `LATIN CAPITAL LETTER Y WITH DIAERESIS`, because 0xFF may not be `LATIN SMALL LETTER Y WITH DIAERESIS` in the current locale, and Perl has no way of knowing if that character even exists in the locale, much less what code point it is.

This modifier may be specified to be the default by `use locale`, but see *Which character set modifier is in effect?*.

/u

means to use Unicode rules when pattern matching. On ASCII platforms, this means that the code points between 128 and 255 take on their Latin-1 (ISO-8859-1) meanings (which are the same as Unicode's). (Otherwise Perl considers their meanings to be undefined.) Thus, under this modifier, the ASCII platform effectively becomes a Unicode platform; and hence, for example, `\w` will match any of the more than 100_000 word characters in Unicode.

Unlike most locales, which are specific to a language and country pair, Unicode classifies all the characters that are letters *somewhere* in the world as `\w`. For example, your locale might not think that `LATIN SMALL LETTER ETH` is a letter (unless you happen to speak Icelandic), but Unicode does. Similarly, all the characters that are decimal digits somewhere in the world will match `\d`; this is hundreds, not 10, possible matches. And some of those digits look like some of the 10 ASCII digits, but mean a different number, so a human could easily think a number is a different quantity than it

really is. For example, BENGALI DIGIT FOUR (U+09EA) looks very much like an ASCII DIGIT EIGHT (U+0038). And, \d+, may match strings of digits that are a mixture from different writing systems, creating a security issue. *"num()" in Unicode::UCD* can be used to sort this out. Or the /a modifier can be used to force \d to match just the ASCII 0 through 9.

Also, under this modifier, case-insensitive matching works on the full set of Unicode characters. The KELVIN SIGN, for example matches the letters "k" and "K"; and LATIN SMALL LIGATURE FF matches the sequence "ff", which, if you're not prepared, might make it look like a hexadecimal constant, presenting another potential security issue. See *http://unicode.org/reports/tr36* for a detailed discussion of Unicode security issues.

This modifier may be specified to be the default by `use feature 'unicode_strings`, `use locale ':not_characters'`, or *`use 5.012`* (or higher), but see *Which character set modifier is in effect?*.

/d

This modifier means to use the "Default" native rules of the platform except when there is cause to use Unicode rules instead, as follows:

1 the target string is encoded in UTF-8; or

2 the pattern is encoded in UTF-8; or

3 the pattern explicitly mentions a code point that is above 255 (say by \x{100}); or

4 the pattern uses a Unicode name (\N{...}); or

5 the pattern uses a Unicode property (\p{...}); or

6 the pattern uses *(?[ ])*

Another mnemonic for this modifier is "Depends", as the rules actually used depend on various things, and as a result you can get unexpected results. See *"The "Unicode Bug"" in perlunicode*. The Unicode Bug has become rather infamous, leading to yet another (printable) name for this modifier, "Dodgy".

Unless the pattern or string are encoded in UTF-8, only ASCII characters can match positively.

Here are some examples of how that works on an ASCII platform:

```
$str =  "\xDF";      # $str is not in UTF-8 format.
$str =~ /^\w/;       # No match, as $str isn't in UTF-8 format.
$str .= "\x{0e0b}";  # Now $str is in UTF-8 format.
$str =~ /^\w/;       # Match! $str is now in UTF-8 format.
chop $str;
$str =~ /^\w/;       # Still a match! $str remains in UTF-8 format.
```

This modifier is automatically selected by default when none of the others are, so yet another name for it is "Default".

Because of the unexpected behaviors associated with this modifier, you probably should only use it to maintain weird backward compatibilities.

/a (and /aa)

This modifier stands for ASCII-restrict (or ASCII-safe). This modifier, unlike the others, may be doubled-up to increase its effect.

When it appears singly, it causes the sequences \d, \s, \w, and the Posix character classes to match only in the ASCII range. They thus revert to their pre-5.6, pre-Unicode meanings. Under /a, \d always means precisely the digits "0" to "9"; \s means the five characters [\f\n\r\t], and

starting in Perl v5.18, experimentally, the vertical tab; `\w` means the 63 characters `[A-Za-z0-9_]`; and likewise, all the Posix classes such as `[[:print:]]` match only the appropriate ASCII-range characters.

This modifier is useful for people who only incidentally use Unicode, and who do not wish to be burdened with its complexities and security concerns.

With `/a`, one can write `\d` with confidence that it will only match ASCII characters, and should the need arise to match beyond ASCII, you can instead use `\p{Digit}` (or `\p{Word}` for `\w`). There are similar `\p{...}` constructs that can match beyond ASCII both white space (see *"Whitespace" in perlrecharclass*), and Posix classes (see *"POSIX Character Classes" in perlrecharclass*). Thus, this modifier doesn't mean you can't use Unicode, it means that to get Unicode matching you must explicitly use a construct (`\p{}`, `\P{}`) that signals Unicode.

As you would expect, this modifier causes, for example, `\D` to mean the same thing as `[^0-9]`; in fact, all non-ASCII characters match `\D`, `\S`, and `\W`. `\b` still means to match at the boundary between `\w` and `\W`, using the `/a` definitions of them (similarly for `\B`).

Otherwise, `/a` behaves like the `/u` modifier, in that case-insensitive matching uses Unicode semantics; for example, "k" will match the Unicode `\N{KELVIN SIGN}` under `/i` matching, and code points in the Latin1 range, above ASCII will have Unicode rules when it comes to case-insensitive matching.

To forbid ASCII/non-ASCII matches (like "k" with `\N{KELVIN SIGN}`), specify the "a" twice, for example `/aai` or `/aia`. (The first occurrence of "a" restricts the `\d`, etc., and the second occurrence adds the `/i` restrictions.) But, note that code points outside the ASCII range will use Unicode rules for `/i` matching, so the modifier doesn't really restrict things to just ASCII; it just forbids the intermixing of ASCII and non-ASCII.

To summarize, this modifier provides protection for applications that don't wish to be exposed to all of Unicode. Specifying it twice gives added protection.

This modifier may be specified to be the default by `use re '/a'` or `use re '/aa'`. If you do so, you may actually have occasion to use the `/u` modifier explictly if there are a few regular expressions where you do want full Unicode rules (but even here, it's best if everything were under feature `"unicode_strings"`, along with the `use re '/aa'`). Also see *Which character set modifier is in effect?*.

Which character set modifier is in effect?

Which of these modifiers is in effect at any given point in a regular expression depends on a fairly complex set of interactions. These have been designed so that in general you don't have to worry about it, but this section gives the gory details. As explained below in *Extended Patterns* it is possible to explicitly specify modifiers that apply only to portions of a regular expression. The innermost always has priority over any outer ones, and one applying to the whole expression has priority over any of the default settings that are described in the remainder of this section.

The *`use re '/foo'`* pragma can be used to set default modifiers (including these) for regular expressions compiled within its scope. This pragma has precedence over the other pragmas listed below that also change the defaults.

Otherwise, *`use locale`* sets the default modifier to `/l`; and *`use feature 'unicode_strings`*, or *`use 5.012`* (or higher) set the default to `/u` when not in the same scope as either *`use locale`* or *`use bytes`*. (*`use locale ':not_characters'`* also sets the default to `/u`, overriding any plain `use locale`.) Unlike the mechanisms mentioned above, these affect operations besides regular expressions pattern matching, and so give more consistent results with other operators, including using `\U`, `\l`, etc. in substitution replacements.

If none of the above apply, for backwards compatibility reasons, the `/d` modifier is the one in effect by default. As this can lead to unexpected results, it is best to specify which other rule set should be

Character set modifier behavior prior to Perl 5.14

Prior to 5.14, there were no explicit modifiers, but `/l` was implied for regexes compiled within the scope of `use locale`, and `/d` was implied otherwise. However, interpolating a regex into a larger regex would ignore the original compilation in favor of whatever was in effect at the time of the second compilation. There were a number of inconsistencies (bugs) with the `/d` modifier, where Unicode rules would be used when inappropriate, and vice versa. `\p{}` did not imply Unicode rules, and neither did all occurrences of `\N{}`, until 5.12.

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
[]       Bracketed 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 option was removed in perl 5.10.)

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 and does not form part of a backslashed sequence like `\x{...}`, it is treated as a regular character. In particular, the lower quantifier bound is not optional, and a typo in a quantifier silently causes it to be treated as the literal characters. For example,

```
/o{4,3}/
```

looks like a quantifier that matches 0 times, since 4 is greater than 3, but it really means to match the sequence of six characters `"o { 4 , 3 }"`. It is planned to eventually require literal uses of curly brackets to be escaped, say by preceding them with a backslash or enclosing them within square brackets, (`"\{"` or `"[{]"`). This change will allow for future syntax extensions (like making the lower

bound of a quantifier optional), and better error checking. In the meantime, you should get in the habit of escaping all instances where you mean a literal "{".)

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 (redundant)
{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 *(?>pattern)* 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)
\cK         control char          (example: VT)
\x{}, \x00  character whose ordinal is the given hexadecimal number
\N{name}    named Unicode character or character sequence
\N{U+263D}  Unicode character     (example: FIRST QUARTER MOON)
\o{}, \000  character whose ordinal is the given octal number
\l          lowercase next char (think vi)
\u          uppercase next char (think vi)
\L          lowercase till \E (think vi)
\U          uppercase till \E (think vi)
\Q          quote (disable) pattern metacharacters till \E
\E          end either case modification or quoted section, think vi
```

Details are in *"Quote and Quote-like Operators" in perlop*.

Character Classes and other Special Escapes

In addition, Perl defines the following:

```
 Sequence   Note    Description
  [...]     [1]  Match a character according to the rules of the
                   bracketed character class defined by the "...".
                   Example: [a-z] matches "a" or "b" or "c" ... or "z"
  [[:...:]] [2]  Match a character according to the rules of the POSIX
                   character class "..." within the outer bracketed
                   character class.  Example: [[:upper:]] matches any
                   uppercase character.
  (?[...])  [8]  Extended bracketed character class
  \w        [3]  Match a "word" character (alphanumeric plus "_", plus
                   other connector punctuation chars plus Unicode
                   marks)
  \W        [3]  Match a non-"word" character
  \s        [3]  Match a whitespace character
  \S        [3]  Match a non-whitespace character
  \d        [3]  Match a decimal digit character
  \D        [3]  Match a non-digit character
  \pP       [3]  Match P, named property.  Use \p{Prop} for longer names
  \PP       [3]  Match non-P
  \X        [4]  Match Unicode "eXtended grapheme cluster"
  \C             Match a single C-language char (octet) even if that is
                   part of a larger UTF-8 character.  Thus it breaks up
                   characters into their UTF-8 bytes, so you may end up
                   with malformed pieces of UTF-8.  Unsupported in
                   lookbehind.
  \1        [5]  Backreference to a specific capture group or buffer.
                   '1' may actually be any positive integer.
  \g1       [5]  Backreference to a specific or previous group,
  \g{-1}    [5]  The number may be negative indicating a relative
                   previous group and may optionally be wrapped in
                   curly brackets for safer parsing.
  \g{name}  [5]  Named backreference
  \k<name>  [5]  Named backreference
  \K        [6]  Keep the stuff left of the \K, don't include it in $&
  \N        [7]  Any character but \n.  Not affected by /s modifier
  \v        [3]  Vertical whitespace
```

```
\V        [3]  Not vertical whitespace
\h        [3]  Horizontal whitespace
\H        [3]  Not horizontal whitespace
\R        [4]  Linebreak
```

[1]

See *"Bracketed Character Classes" in perlrecharclass* for details.

[2]

See *"POSIX Character Classes" in perlrecharclass* for details.

[3]

See *"Backslash sequences" in perlrecharclass* for details.

[4]

See *"Misc" in perlrebackslash* for details.

[5]

See *Capture groups* below for details.

[6]

See *Extended Patterns* below for details.

[7]

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

[8]

See *"Extended Bracketed Character Classes" in perlrecharclass* for details.

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 (see *Repeated Patterns*

Matching a Zero-length Substring) is modified somewhat, in that contents to the left of `\G` are not counted when determining the length of the match. Thus the following will not match forever:

```
my $string = 'ABC';
pos($string) = 1;
while ($string =~ /(.\G)/g) {
    print $1;
}
```

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 groups

The bracketing construct `( ... )` creates capture groups (also referred to as capture buffers). To refer to the current contents of a group later on, within the same pattern, use `\g1` (or `\g{1}`) for the first, `\g2` (or `\g{2}`) for the second, and so on. This is called a *backreference*. There is no limit to the number of captured substrings that you may use. Groups are numbered with the leftmost open parenthesis being number 1, etc. If a group did not match, the associated backreference won't match either. (This can happen if the group is optional, or in a different branch of an alternation.) You can omit the `"g"`, and write `"\1"`, etc, but there are some issues with this form, described below.

You can also refer to capture groups relatively, by using a negative number, so that `\g-1` and `\g{-1}` both refer to the immediately preceding capture group, and `\g-2` and `\g{-2}` both refer to the group before it. For example:

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

would match the same as `/(Y) ( (X) \g3 \g1 )/x`. This allows you to interpolate regexes into larger regexes and not have to worry about the capture groups being renumbered.

You can dispense with numbers altogether and create named capture groups. The notation is `(?<`*name*`>...)` to declare and `\g{`*name*`}` to reference. (To be compatible with .Net regular expressions, `\g{`*name*`}` may also be written as `\k{`*name*`}`, `\k<`*name*`>` or `\k'`*name*`'`.) *name* must not begin with a number, nor contain hyphens. When different groups within the same pattern have the same name, any reference to that name assumes the leftmost defined group. Named groups count in absolute and relative numbering, and so can also be referred to by those numbers. (It's possible to do things with named capture groups that would otherwise require `(??{})`.)

Capture group contents are dynamically scoped and available to you outside the pattern until the end of the enclosing block or until the next successful match, whichever comes first. (See *"Compound Statements" in perlsyn*.) You can refer to them by absolute number (using `"$1"` instead of `"\g1"`, etc); or by name via the `%+` hash, using `"$+{`*name*`}"`.

Braces are required in referring to named capture groups, but are optional for absolute or relative numbered ones. Braces are safer when creating a regex by concatenating smaller strings. For example if you have `qr/$a$b/`, and `$a` contained `"\g1"`, and `$b` contained `"37"`, you would get `/\g137/` which is probably not what you intended.

The \g and \k notations were introduced in Perl 5.10.0. Prior to that there were no named nor relative numbered capture groups. Absolute numbered groups were referred to using \1, \2, etc., and this notation is still accepted (and likely always will be). But it leads to some ambiguities if there are more than 9 capture groups, as \10 could mean either the tenth capture group, or the character whose ordinal in octal is 010 (a backspace in 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. There are several examples below that illustrate these perils. You can avoid the ambiguity by always using \g{} or \g if you mean capturing groups; and for octal constants always using \o{}, or for \077 and below, using 3 digits padded with leading zeros, since a leading zero implies an octal constant.

The *digit* notation also works in certain circumstances outside the pattern. See *Warning on \1 Instead of $1* below for details.

Examples:

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

/(.)\g1/                        # 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'.)\g1/                 # ... mix and match
     and print "'$1' is the first doubled character\n";

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

/(.)(.)(.)(.)(.)(.)(.)(.)(.)\g10/   # \g10 is a backreference
/(.)(.)(.)(.)(.)(.)(.)(.)(.)\10/    # \10 is octal
/((.)(.)(.)(.)(.)(.)(.)(.)(.))\10/  # \10 is a backreference
/((.)(.)(.)(.)(.)(.)(.)(.)(.))\010/ # \010 is octal

$a = '(.)\1';        # Creates problems when concatenated.
$b = '(.)\g{1}';     # Avoids the problems.
"aa" =~ /${a}/;      # True
"aa" =~ /${b}/;      # True
"aa0" =~ /${a}0/;    # False!
"aa0" =~ /${b}0/;    # True
"aa\x08" =~ /${a}0/;  # True!
"aa\x08" =~ /${b}0/;  # False
```

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.

These special variables, like the %+ hash and the numbered match variables ($1, $2, $3, etc.) are 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 5.17.4, the presence of each of the three variables in a program is recorded separately, and depending on circumstances, perl may be able be more efficient knowing that only $& rather than all three have been seen, for example.

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.

Quoting metacharacters

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*.

quotemeta() and \Q are fully described in *"quotemeta" in perlfunc*.

Extended Patterns

Perl also defines a consistent extension syntax for features not found in standard tools like **awk** and **lex**. The syntax for most of these 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.

`(?adlupimsx-imsx)`

`(?^alupimsx)`

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+ \g1
```

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.

Any of these modifiers can be set to apply globally to all regular expressions compiled within the scope of a `use re`. See *"'/flags' mode" in re*.

Starting in Perl 5.14, a `"^"` (caret or circumflex accent) immediately after the `"?"` is a shorthand equivalent to `d-imsx`. Flags (except `"d"`) may follow the caret to override it. But a minus sign is not legal with it.

Note that the `a`, `d`, `l`, `p`, and `u` modifiers are special in that they can only be enabled, not disabled, and the `a`, `d`, `l`, and `u` modifiers are mutually exclusive: specifying one de-specifies the others, and a maximum of one (or two a's) may appear in the construct. Thus, for example, `(?-p)` will warn when compiled under `use warnings`; `(?-d:...)` and `(?dl:...)` are fatal errors.

Note also that the `p` modifier is special in that its presence anywhere in a pattern has a global effect.

`(?:pattern)`

`(?adluimsx-imsx:pattern)`

`(?^aluimsx: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 (?adluimsx-imsx). For example,

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

is equivalent to the more verbose

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

Starting in Perl 5.14, a "^" (caret or circumflex accent) immediately after the "?" is a shorthand equivalent to d-imsx. Any positive flags (except "d") may follow the caret, so

```
(?^x:foo)
```

is equivalent to

```
(?x-ims:foo)
```

The caret tells Perl that this cluster doesn't inherit the flags of any surrounding pattern, but uses the system defaults (d-imsx), modified by any flags specified.

The caret allows for simpler stringification of compiled regular expressions. These look like

```
(?^:pattern)
```

with any non-default flags appearing between the caret and the colon. A test that looks at such stringification thus doesn't need to have the system default flags hard-coded in it, just the caret. If new flags are added to Perl, the meaning of the caret's expansion will change to include the default for those flags, so the test will still work, unchanged.

Specifying a negative flag after the caret is an error, as the flag is redundant.

Mnemonic for (?^...): A fresh beginning since the usual use of a caret is to match at the beginning.

(?|pattern)

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

Capture groups 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 groups following this construct will be numbered as though the construct contained only one branch, that being the one with the most capture groups in it.

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

Consider the following pattern. The numbers underneath show in which group 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 groups 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 group named a and the group named b are aliases for the group 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. Use look-behind instead (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 group. Identical in every respect to normal capturing parentheses `()` but for the additional fact that the group can be referred to by name in various regular expression constructs (like `\g{NAME}`) and can be accessed by name after a successful match via `%+` or

`%-`. See *perlvar* for more details on the `%+` and `%-` hashes.

If multiple distinct capture groups have the same name then the $+{NAME} will refer to the leftmost defined group 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 groups 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. The implementation of this feature was radically overhauled for the 5.18.0 release, and its behaviour in earlier versions of perl was much buggier, especially in relation to parsing, lexical vars, scoping, recursion and reentrancy.

This zero-width assertion executes any embedded Perl code. It always succeeds, and its return value is set as `$^R`.

In literal patterns, the code is parsed at the same time as the surrounding code. While within the pattern, control is passed temporarily back to the perl parser, until the logically-balancing closing brace is encountered. This is similar to the way that an array index expression in a literal string is handled, for example

```
"abc$array[ 1 + f('[') + g()]def"
```

In particular, braces do not need to be balanced:

```
s/abc(?{ f('{'); })/def/
```

Even in a pattern that is interpolated and compiled at run-time, literal code blocks will be compiled once, at perl compile time; the following prints "ABCD":

```
print "D";
my $qr = qr/(?{ BEGIN { print "A" } })/;
```

```
my $foo = "foo";
/$foo$qr(?{ BEGIN { print "B" } })/;
BEGIN { print "C" }
```

In patterns where the text of the code is derived from run-time information rather than appearing literally in a source code /pattern/, the code is compiled at the same time that the pattern is compiled, and for reasons of security, `use re 'eval'` must be in scope. This is to stop user-supplied patterns containing code snippets from being executable.

In situations where you need to enable this with `use re 'eval'`, you should also have taint checking enabled. Better yet, use the carefully constrained evaluation within a Safe compartment. See *perlsec* for details about both these mechanisms.

From the viewpoint of parsing, lexical variable scope and closures,

```
/AAA(?{ BBB })CCC/
```

behaves approximately like

```
/AAA/ && do { BBB } && /CCC/
```

Similarly,

```
qr/AAA(?{ BBB })CCC/
```

behaves approximately like

```
sub { /AAA/ && do { BBB } && /CCC/ }
```

In particular:

```
{ my $i = 1; $r = qr/(?{ print $i })/ }
my $i = 2;
/$r/; # prints "1"
```

Inside a `(?{...})` 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 block introduces a new scope from the perspective of lexical variable declarations, but **not** from the perspective of `local` and similar localizing behaviours. So later code blocks within the same pattern will still see the values which were localized in earlier blocks. These accumulated localizations are undone either at the end of a successful match, or if the assertion is backtracked (compare *Backtracking*). For example,

```
$_ = '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 initially increment `$cnt` up to 8; then during backtracking, its value will be unwound back to 4, which is the value assigned to `$res`. At the end of the regex execution, $cnt will be wound back to its initial value of 0.

This assertion may be used as the condition in 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*.

Note that the special variable `$^N` is particularly useful with code blocks 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";
```

`(??{ 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. It behaves in *exactly* the same way as a `(?{ code })` code block as described above, except that its return value, rather than being assigned to `$^R`, is treated as a pattern, compiled if it's a string (or used as-is if its a qr// object), then matched as if it were inserted instead of this construct.

During the matching of this sub-pattern, it has its own set of captures which are valid during the sub-match, but are discarded once control returns to the main pattern. For example, the following matches, with the inner pattern capturing "B" and matching "BB", while the outer pattern captures "A";

```
my $inner = '(.)\1';
"ABBA" =~ /^(.)(??{ $inner })\1/;
print $1; # prints "A";
```

Note that this means that there is no way for the inner pattern to refer to a capture group defined outside. (The code block itself can use `$1`, etc., to refer to the enclosing pattern's capture groups.) Thus, although

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

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

The following pattern matches a parenthesized group:

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

See also *(?PARN0)* for a different, more efficient way to accomplish the same task.

Executing a postponed regular expression 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 that it does not involve executing any code or potentially compiling a returned pattern string; instead it treats the part of the current pattern contained within a specified capture group as an independent pattern that must match at the current position. Capture groups 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 group 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 groups and positive ones following. Thus `(?-1)` refers to the most recently declared group, and `(?+1)` indicates the next group to be declared. Note that the counting for relative recursion differs from that of relative backreferences, in that with recursion unclosed groups **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 group 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. Matches `yes-pattern` if `condition` yields a true value, matches `no-pattern` otherwise. A missing pattern always matches.

`(condition)` should be one of: 1) an integer in parentheses (which is valid if the corresponding pair of parentheses matched); 2) a look-ahead/look-behind/evaluate zero-width assertion; 3) a name in angle brackets or single quotes (which is valid if a group with the given name matched); or 4) 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 group has matched something.

(<NAME>) ('NAME')

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

(?=...) (?!...) (?<=...) (?<!...)

Checks whether the pattern matches (or does not match, for the '!' variants).

(?{ CODE })

Treats the return value of 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 its yes-pattern directly, and does not allow a no-pattern. This allows one to define subpatterns which will be executed only by 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 groups matched inside of recursion are not accessible after the recursion returns, so the extra layer of capturing groups is necessary. Thus `$+{NAME_PAT}` would not be defined even though `$+{NAME}` would be.

Finally, keep in mind that subpatterns created inside a DEFINE block count towards the absolute and relative number of captures, so this:

```
my @captures = "a" =~ /(.)                  # First capture
                       (?(DEFINE)
                           (?<EXAMPLE> 1 )  # Second capture
                       )/x;
say scalar @captures;
```

Will output 2, not 1. This is particularly important if you intend to compile the definitions with the `qr//` operator, and later interpolate them in another pattern.

`(?>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.

`(?>pattern)` does not disable backtracking altogether once it has matched. It is still possible to backtrack past the construct, but not into it. So `((?>a*)|(?>b*))ar` will still match "bar".

An effect similar to `(?>pattern)` may be achieved by writing `(?=(pattern))\g{-1}`. This matches the same substring as a standalone `a+`, and the following `\g{-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, when this construct is followed by a quantifier, it 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})
```

`(?[ ])`

See *"Extended Bracketed Character Classes" in perlrecharclass*.

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 string 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)`

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 group 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) that has alternations. The two branches of a `(?(condition)yes-pattern|no-pattern)` do not count as an alternation, as far as `(*THEN)` is concerned.

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 ) /
```

is not the same as

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

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.

Verbs without an argument

`(*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.

`(*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 groups then the groups 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 was 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 three octal digits, matches the character whose coded character set value is *nnn*. Similarly, \x*nn*, where *nn* are hexadecimal digits, matches the character whose ordinal 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 ("(", "(?:", etc. or the beginning of the pattern) up to the first "|", and the last alternative contains everything from the last "|" to the next closing 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* or \g*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\g1\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 \1 to \9) 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;
```

For example, this program

```
#!perl -l
"aaaaab" =~ /
  (?:
     a                # non-zero
     |                # or
    (?{print "hello"}) # print hello whenever this
                       #    branch is tried
    (?=(b))           # zero-width assertion
  )*  # any number of times
 /x;
print $&;
```

```
print $1;
```

prints

```
hello
aaaaa
b
```

Notice that "hello" is only printed once, as when Perl sees that the sixth iteration of the outermost `(?:)*` matches a zero-length string, it stops the `*`.

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 a 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 a 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 group *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

As of Perl 5.10.0, one can create custom regular expression engines. This is not for the faint of heart, as they have to plug in at the C level. See *perlreapi* for more details.

As an alternative, overloaded constants (see *overload*) provide a simple way to extend the functionality of the RE engine, by substituting one pattern for another.

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 group. Equivalent to `(?<NAME>pattern)`.

`(?P=NAME)`

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

`(?P>NAME)`

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

BUGS

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

There are a number of issues with regard to case-insensitive matching in Unicode rules. See `i` under *Modifiers* above.

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.