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*.

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*.

i

Do case-insensitive pattern matching.

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

m

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

s

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

The `/s` and `/m` modifiers both override the `$*` setting. That is, no matter what `$*` contains, `/s` without `/m` will force "^" to match only at the beginning of the string and "$" to match only at the end (or just before a newline at the end) of the string. Together, as /ms, they let the "." match any character whatsoever, while still allowing "^" and "$" to match, respectively, just after and just before newlines within the string.

x

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

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

The `/x` modifier itself needs a little more explanation. It tells the regular expression parser to ignore 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`), that you'll either have to escape them or encode them using octal or hex 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*.

Regular Expressions

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

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

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

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

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. The `/s` modifier also overrides the setting of `$*`, in case you have some (badly behaved) older code that sets it in another module.

The following standard quantifiers are recognized:

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

(If a curly bracket occurs in any other context, it is treated as a regular character. In particular, the lower bound is not optional.) The "*" modifier is equivalent to `{0,}`, the "+" modifier to `{1,}`, and the "?" modifier to `{0,1}`. n and m are limited to 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
+?    Match 1 or more times
??    Match 0 or 1 time
{n}?   Match exactly n times
{n,}?  Match at least n times
{n,m}? Match at least n but not more than m times
```

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

```
\t  tab                   (HT, TAB)
\n  newline               (LF, NL)
\r  return                (CR)
\f  form feed             (FF)
\a  alarm (bell)          (BEL)
```

```
    \e  escape (think troff)  (ESC)
    \033 octal char (think of a PDP-11)
    \x1B hex char
    \x{263a} wide hex char         (Unicode SMILEY)
    \c[  control char
    \N{name} named char
    \l  lowercase next char (think vi)
    \u  uppercase next char (think vi)
    \L  lowercase till \E (think vi)
    \U  uppercase till \E (think vi)
    \E  end case modification (think vi)
    \Q  quote (disable) pattern metacharacters till \E
```

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

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

In addition, Perl defines the following:

```
    \w Match a "word" character (alphanumeric plus "_")
    \W Match a non-"word" character
    \s Match a whitespace character
    \S Match a non-whitespace character
    \d Match a digit character
    \D Match a non-digit character
    \pP Match P, named property.  Use \p{Prop} for longer names.
    \PP Match non-P
    \X Match eXtended Unicode "combining character sequence",
        equivalent to (?:\PM\pM*)
    \C Match a single C char (octet) even under Unicode.
 NOTE: breaks up characters into their UTF-8 bytes,
 so you may end up with malformed pieces of UTF-8.
 Unsupported in lookbehind.
```

A `\w` matches a single alphanumeric character (an alphabetic character, or a decimal digit) or `_`, not a whole word. Use `\w+` to match a string of Perl-identifier characters (which isn't the same as matching an English word). If `use locale` is in effect, the list of alphabetic characters generated by `\w` is taken from the current locale. See *perllocale*. You may use `\w`, `\W`, `\s`, `\S`, `\d`, and `\D` within character classes, but if you try to use them as endpoints of a range, that's not a range, the "-" is understood literally. If Unicode is in effect, `\s` matches also "\x{85}", "\x{2028}, and "\x{2029}", see *perlunicode* for more details about `\pP`, `\PP`, and `\X`, and *perluniintro* about Unicode in general. You can define your own `\p` and `\P` properties, see *perlunicode*.

The POSIX character class syntax

```
    [:class:]
```

is also available. The available classes and their backslash equivalents (if available) are as follows:

```
    alpha
    alnum
    ascii
    blank  [1]
```

```
cntrl
digit       \d
graph
lower
print
punct
space       \s [2]
upper
word        \w [3]
xdigit
```

[1]

A GNU extension equivalent to `[ \t]`, "all horizontal whitespace".

[2]

Not exactly equivalent to `\s` since the `[[:space:]]` includes also the (very rare) "vertical tabulator", "\ck", chr(11).

[3]

A Perl extension, see above.

For example use `[:upper:]` to match all the uppercase characters. Note that the `[]` are part of the `[::]` construct, not part of the whole character class. For example:

```
[01[:alpha:]%]
```

matches zero, one, any alphabetic character, and the percentage sign.

The following equivalences to Unicode \p{} constructs and equivalent backslash character classes (if available), will hold:

```
[:...:] \p{...}  backslash

alpha       IsAlpha
alnum       IsAlnum
ascii       IsASCII
blank       IsSpace
cntrl       IsCntrl
digit       IsDigit        \d
graph       IsGraph
lower       IsLower
print       IsPrint
punct       IsPunct
space       IsSpace
            IsSpacePerl    \s
upper       IsUpper
word        IsWord
xdigit      IsXDigit
```

For example `[:lower:]` and `\p{IsLower}` are equivalent.

If the `utf8` pragma is not used but the `locale` pragma is, the classes correlate with the usual isalpha(3) interface (except for "word" and "blank").

The assumedly non-obviously named classes are:

cntrl

Any control character. Usually characters that don't produce output as such but instead control the terminal somehow: for example newline and backspace are control characters. All characters with ord() less than 32 are most often classified as control characters (assuming ASCII, the ISO Latin character sets, and Unicode), as is the character with the ord() value of 127 (DEL).

graph

Any alphanumeric or punctuation (special) character.

print

Any alphanumeric or punctuation (special) character or the space character.

punct

Any punctuation (special) character.

xdigit

Any hexadecimal digit. Though this may feel silly ([0-9A-Fa-f] would work just fine) it is included for completeness.

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

```
POSIX traditional Unicode

[:^digit:]      \D      \P{IsDigit}
[:^space:]     \S     \P{IsSpace}
[:^word:]     \W     \P{IsWord}
```

Perl respects the POSIX standard in that POSIX character classes are only supported within a character class. The POSIX character classes [.cc.] and [=cc=] are recognized but **not** supported and trying to use them will cause an error.

Perl defines the following zero-width assertions:

```
\b Match a word boundary
\B Match a non-(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*. Currently \G is only fully supported when anchored to the start of the pattern; while it is permitted to use it elsewhere, as in /(?<=\G..)./g, some such uses (/.\G/g, for example) currently cause problems, and it is recommended that you avoid such usage

for now.

The bracketing construct (...) creates capture buffers. To refer to the digit'th buffer use \<digit> within the match. Outside the match use "$" instead of "\". (The \<digit> notation works in certain circumstances outside the match. See the warning below about \1 vs $1 for details.) Referring back to another part of the match is called a *backreference*.

There is no limit to the number of captured substrings that you may use. However Perl also uses \10, \11, etc. as aliases for \010, \011, etc. (Recall that 0 means octal, so \011 is the character at number 9 in your coded character set; which would be the 10th character, a horizontal tab under ASCII.) Perl resolves this ambiguity by interpreting \10 as a backreference only if at least 10 left parentheses have opened before it. Likewise \11 is a backreference only if at least 11 left parentheses have opened before it. And so on. \1 through \9 are always interpreted as backreferences.

Examples:

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

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

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

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

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

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

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

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

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

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

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

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

Extended Patterns

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

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

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

`(?#text)`

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

`(?imsx-imsx)`

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

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

# more flexible:

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

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

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

will match a repeated (*including the case*!) word `blah` in any case, assuming `x` modifier, and no `i` modifier outside this group.

`(?:pattern)`

`(?imsx-imsx:pattern)`

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

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

is like

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

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

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

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

is equivalent to the more verbose

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

`(?=pattern)`

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

`(?!pattern)`

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

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

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

For look-behind see below.

`(?<=pattern)`

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.

`(?<!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.

`(?{ code })`

WARNING: This extended regular expression feature is considered highly experimental, and may be changed or deleted without notice.

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

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

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

```
  print "color = $color, animal = $animal\n";
```

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

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

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

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

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

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

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

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

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

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

`(??{ code })`

WARNING: This extended regular expression feature is considered highly experimental, and may be changed or deleted without notice. A simplified version of the syntax may be introduced for commonly used idioms.

This is a "postponed" regular subexpression. The `code` is evaluated at run time, at

the moment this subexpression may match. The result of evaluation is considered as a regular expression and matched as if it were inserted instead of this construct.

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

The following pattern matches a parenthesized group:

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

`(?>pattern)`

WARNING: This extended regular expression feature is considered highly experimental, and may be changed or deleted without notice.

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

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

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

Consider this pattern:

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

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

program has hung. However, a tiny change to this pattern

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

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

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

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

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

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

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

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

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

`(?(condition)yes-pattern)`

WARNING: This extended regular expression feature is considered highly experimental, and may be changed or deleted without notice.

Conditional expression. `(condition)` should be either an integer in parentheses (which is valid if the corresponding pair of parentheses matched), or look-ahead/look-behind/evaluate zero-width assertion.

For example:

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

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

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 pieces together*.

A fundamental feature of regular expression matching involves the notion called *backtracking*, which is currently used (when needed) by all regular 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 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 "\"). 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 one 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 coded character sets.) Also, if you try to use the character classes `\w`, `\W`, `\s`, `\S`, `\d`, or `\D` as endpoints of a range, that's not 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 alphabets of equal case ([a-e], [A-E]), or digits ([0-9]). Anything else is unsafe. If in doubt, spell out the character sets in full.

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

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

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

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

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

"0x1234 01234", because subpattern 1 matched "0x", even though the rule `0|0x` could potentially match the leading 0 in the second number.

Warning on \1 vs $1

Some people get too used to writing things like:

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

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

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

Or if you try to do

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

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

Repeated patterns matching 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?` can match 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 `*` modifier. 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 modifiers `*+{}`, and for higher-level ones like the `/g` modifier or split() operator.

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

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

is made equivalent to

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

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

For example:

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

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

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

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

Combining pieces together

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

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

Another description starts with notions of "better"/"worse". All the substrings which may be matched by the given regular expression can be sorted from the "best" match to the "worst" match, and it is the "best" match which is chosen. This substitutes the question of "what is chosen?" by the question of "which matches are better, and which are worse?".

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

`ST`

> Consider two possible matches, `AB` and `A'B'`, `A` and `A'` are substrings which can be matched by `S`, `B` and `B'` are substrings which can be matched by `T`.
>
> If `A` is better match for `S` than `A'`, `AB` is a better match than `A'B'`.

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

S|T

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

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

S{REPEAT_COUNT}

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

S{min,max}

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

S{min,max}?

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

S?, S*, S+

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

S??, S*?, S+?

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

(?>S)

Matches the best match for S and only that.

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

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

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

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

(??{ EXPR })

The ordering is the same as for the regular expression which is the result of EXPR.

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

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

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

Creating custom RE engines

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

Suppose that we want to enable a new RE escape-sequence \Y| which matches at 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|/;
```

BUGS

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.