NAME

re - Perl pragma to alter regular expression behaviour

SYNOPSIS

```
    use re 'taint';
    ($x) = ($^X =~ /^(.*)$/s);     # $x is tainted here

    $pat = '(?{ $foo = 1 })';
    use re 'eval';
    /foo${pat}bar/;     # won't fail (when not under -T
                                   # switch)

    {
no re 'taint';     # the default
($x) = ($^X =~ /^(.*)$/s); # $x is not tainted here

no re 'eval';     # the default
/foo${pat}bar/;     # disallowed (with or without -T
                                   # switch)
    }

    use re 'strict';               # Raise warnings for more conditions

    use re '/ix';
    "FOO" =~ / foo /; # /ix implied
    no re '/x';
    "FOO" =~ /foo/; # just /i implied

    use re 'debug';     # output debugging info during
    /^(.*)$/s;      # compile and run time

    use re 'debugcolor';    # same as 'debug', but with colored
                                   # output
    ...

    use re qw(Debug All);          # Same as "use re 'debug'", but you
                                   # can use "Debug" with things other
                                   # than 'All'
    use re qw(Debug More);         # 'All' plus output more details
    no re qw(Debug ALL);           # Turn on (almost) all re debugging
                                   # in this scope

    use re qw(is_regexp regexp_pattern); # import utility functions
    my ($pat,$mods)=regexp_pattern(qr/foo/i);
    if (is_regexp($obj)) {
        print "Got regexp: ",
            scalar regexp_pattern($obj); # just as perl would stringify
    }                                    # it but no hassle with blessed
                                         # re's.
```

(We use $^X in these examples because it's tainted by default.)

DESCRIPTION

'taint' mode

When `use re 'taint'` is in effect, and a tainted string is the target of a regexp, the regexp memories (or values returned by the m// operator in list context) are tainted. This feature is useful when regexp operations on tainted data aren't meant to extract safe substrings, but to perform other transformations.

'eval' mode

When `use re 'eval'` is in effect, a regexp is allowed to contain `(?{ ... })` zero-width assertions and `(??{ ... })` postponed subexpressions that are derived from variable interpolation, rather than appearing literally within the regexp. That is normally disallowed, since it is a potential security risk. Note that this pragma is ignored when the regular expression is obtained from tainted data, i.e. evaluation is always disallowed with tainted regular expressions. See *"(?{ code })" in perlre* and *"(??{ code })" in perlre*.

For the purpose of this pragma, interpolation of precompiled regular expressions (i.e., the result of `qr//`) is *not* considered variable interpolation. Thus:

```
/foo${pat}bar/
```

is allowed if $pat is a precompiled regular expression, even if $pat contains `(?{ ... })` assertions or `(??{ ... })` subexpressions.

'strict' mode

Note that this is an experimental feature which may be changed or removed in a future Perl release.

When `use re 'strict'` is in effect, stricter checks are applied than otherwise when compiling regular expressions patterns. These may cause more warnings to be raised than otherwise, and more things to be fatal instead of just warnings. The purpose of this is to find and report at compile time some things, which may be legal, but have a reasonable possibility of not being the programmer's actual intent. This automatically turns on the `"regexp"` warnings category (if not already on) within its scope.

As an example of something that is caught under `"strict'`, but not otherwise, is the pattern

```
qr/\xABC/
```

The `"\x"` construct without curly braces should be followed by exactly two hex digits; this one is followed by three. This currently evaluates as equivalent to

```
qr/\x{AB}C/
```

that is, the character whose code point value is `0xAB`, followed by the letter `C`. But since `C` is a a hex digit, there is a reasonable chance that the intent was

```
qr/\x{ABC}/
```

that is the single character at `0xABC`. Under `'strict'` it is an error to not follow `\x` with exactly two hex digits. When not under `'strict'` a warning is generated if there is only one hex digit, and no warning is raised if there are more than two.

It is expected that what exactly `'strict'` does will evolve over time as we gain experience with it. This means that programs that compile under it in today's Perl may not compile, or may have more or fewer warnings, in future Perls. There is no backwards compatibility promises with regards to it. Also there are already proposals for an alternate syntax for enabling it. For these reasons, using it will raise a `experimental::re_strict` class warning, unless that category is turned off.

Note that if a pattern compiled within `'strict'` is recompiled, say by interpolating into another pattern, outside of `'strict'`, it is not checked again for strictness. This is because if it works under strict it must work under non-strict.

'/flags' mode

When `use re '/flags'` is specified, the given *flags* are automatically added to every regular expression till the end of the lexical scope. *flags* can be any combination of `'a'`, `'aa'`, `'d'`, `'i'`, `'l'`, `'m'`, `'n'`, `'p'`, `'s'`, `'u'`, `'x'`, and/or `'xx'`.

`no re '/flags'` will turn off the effect of `use re '/flags'` for the given flags.

For example, if you want all your regular expressions to have /msxx on by default, simply put

```
use re '/msxx';
```

at the top of your code.

The character set `/adul` flags cancel each other out. So, in this example,

```
use re "/u";
"ss" =~ /\xdf/;
use re "/d";
"ss" =~ /\xdf/;
```

the second `use re` does an implicit `no re '/u'`.

Similarly,

```
use re "/xx";   # Doubled-x
...
use re "/x";    # Single x from here on
...
```

Turning on one of the character set flags with `use re` takes precedence over the `locale` pragma and the 'unicode_strings' `feature`, for regular expressions. Turning off one of these flags when it is active reverts to the behaviour specified by whatever other pragmata are in scope. For example:

```
use feature "unicode_strings";
no re "/u"; # does nothing
use re "/l";
no re "/l"; # reverts to unicode_strings behaviour
```

'debug' mode

When `use re 'debug'` is in effect, perl emits debugging messages when compiling and using regular expressions. The output is the same as that obtained by running a `-DDEBUGGING`-enabled perl interpreter with the **-Dr** switch. It may be quite voluminous depending on the complexity of the match. Using `debugcolor` instead of `debug` enables a form of output that can be used to get a colorful display on terminals that understand termcap color sequences. Set `$ENV{PERL_RE_TC}` to a comma-separated list of `termcap` properties to use for highlighting strings on/off, pre-point part on/off. See *"Debugging Regular Expressions" in perldebug* for additional info.

As of 5.9.5 the directive `use re 'debug'` and its equivalents are lexically scoped, as the other directives are. However they have both compile-time and run-time effects.

See *"Pragmatic Modules" in perlmodlib*.

'Debug' mode

Similarly `use re 'Debug'` produces debugging output, the difference being that it allows the fine tuning of what debugging output will be emitted. Options are divided into three groups, those related to compilation, those related to execution and those related to special purposes. The options are as follows:

Compile related options

COMPILE

Turns on all compile related debug options.

PARSE

Turns on debug output related to the process of parsing the pattern.

OPTIMISE

Enables output related to the optimisation phase of compilation.

TRIEC

Detailed info about trie compilation.

DUMP

Dump the final program out after it is compiled and optimised.

FLAGS

Dump the flags associated with the program

TEST

Print output intended for testing the internals of the compile process

Execute related options

EXECUTE

Turns on all execute related debug options.

MATCH

Turns on debugging of the main matching loop.

TRIEE

Extra debugging of how tries execute.

INTUIT

Enable debugging of start-point optimisations.

Extra debugging options

EXTRA

Turns on all "extra" debugging options.

BUFFERS

Enable debugging the capture group storage during match. Warning, this can potentially produce extremely large output.

TRIEM

Enable enhanced TRIE debugging. Enhances both TRIEE and TRIEC.

STATE

Enable debugging of states in the engine.

STACK

Enable debugging of the recursion stack in the engine. Enabling or disabling this option automatically does the same for debugging states as well. This output from this can be quite large.

GPOS

Enable debugging of the \G modifier.

OPTIMISEM

Enable enhanced optimisation debugging and start-point optimisations. Probably not useful except when debugging the regexp engine itself.

OFFSETS

Dump offset information. This can be used to see how regops correlate to the pattern. Output format is

```
NODENUM:POSITION[LENGTH]
```

Where 1 is the position of the first char in the string. Note that position can be 0, or larger than the actual length of the pattern, likewise length can be zero.

OFFSETSDBG

Enable debugging of offsets information. This emits copious amounts of trace information and doesn't mesh well with other debug options.

Almost definitely only useful to people hacking on the offsets part of the debug engine.

Other useful flags

These are useful shortcuts to save on the typing.

ALL

Enable all options at once except OFFSETS, OFFSETSDBG and BUFFERS. (To get every single option without exception, use both ALL and EXTRA.)

All

Enable DUMP and all execute options. Equivalent to:

```
use re 'debug';
```

MORE

More

Enable the options enabled by "All", plus STATE, TRIEC, and TRIEM.

As of 5.9.5 the directive `use re 'debug'` and its equivalents are lexically scoped, as are the other directives. However they have both compile-time and run-time effects.

Exportable Functions

As of perl 5.9.5 're' debug contains a number of utility functions that may be optionally exported into the caller's namespace. They are listed below.

is_regexp($ref)

Returns true if the argument is a compiled regular expression as returned by `qr//`, false if it is not.

This function will not be confused by overloading or blessing. In internals terms, this extracts the regexp pointer out of the PERL_MAGIC_qr structure so it cannot be fooled.

regexp_pattern($ref)

If the argument is a compiled regular expression as returned by `qr//`, then this function returns the pattern.

In list context it returns a two element list, the first element containing the pattern and the second containing the modifiers used when the pattern was compiled.

```
my ($pat, $mods) = regexp_pattern($ref);
```

In scalar context it returns the same as perl would when stringifying a raw `qr//` with the same pattern inside. If the argument is not a compiled reference then this routine returns false but defined in scalar context, and the empty list in list context. Thus the following

```
if (regexp_pattern($ref) eq '(?^i:foo)')
```

will be warning free regardless of what $ref actually is.

Like `is_regexp` this function will not be confused by overloading or blessing of the object.

regmust($ref)

If the argument is a compiled regular expression as returned by `qr//`, then this function returns what the optimiser considers to be the longest anchored fixed string and longest floating fixed string in the pattern.

A *fixed string* is defined as being a substring that must appear for the pattern to match. An *anchored fixed string* is a fixed string that must appear at a particular offset from the beginning of the match. A *floating fixed string* is defined as a fixed string that can appear at any point in a range of positions relative to the start of the match. For example,

```
my $qr = qr/here .* there/x;
my ($anchored, $floating) = regmust($qr);
print "anchored:'$anchored'\nfloating:'$floating'\n";
```

results in

```
anchored:'here'
floating:'there'
```

Because the `here` is before the `.*` in the pattern, its position can be determined exactly. That's not true, however, for the `there`; it could appear at any point after where the anchored string appeared. Perl uses both for its optimisations, preferring the longer, or, if they are equal, the floating.

NOTE: This may not necessarily be the definitive longest anchored and floating string. This will be what the optimiser of the Perl that you are using thinks is the longest. If you believe that the result is wrong please report it via the *perlbug* utility.

regname($name,$all)

Returns the contents of a named buffer of the last successful match. If $all is true, then returns an array ref containing one entry per buffer, otherwise returns the first defined buffer.

regnames($all)

Returns a list of all of the named buffers defined in the last successful match. If $all is true, then it returns all names defined, if not it returns only names which were involved in the match.

regnames_count()

Returns the number of distinct names defined in the pattern used for the last successful match.

Note: this result is always the actual number of distinct named buffers defined, it may not actually match that which is returned by `regnames()` and related routines when those routines have not been called with the $all parameter set.

SEE ALSO

"Pragmatic Modules" in perlmodlib.