## NAME

Regexp::Assemble – Assemble multiple Regular Expressions into a single RE

## SYNOPSIS

```
use Regexp::Assemble;

my $ra = Regexp::Assemble->new;
$ra->add( 'ab+c' );
$ra->add( 'ab+-' );
$ra->add( 'a\w\d+' );
$ra->add( 'a\d+' );
print $ra->re; # prints a(?:\w?\d+|b+[-c])
```

## DESCRIPTION

Regexp::Assemble takes an arbitrary number of regular expressions and assembles them into a single regular expression (or RE) that matches all that the individual REs match.

As a result, instead of having a large list of expressions to loop over, a target string only needs to be tested against one expression. This is interesting when you have several thousand patterns to deal with. Serious effort is made to produce the smallest pattern possible.

It is also possible to track the original patterns, so that you can determine which, among the source patterns that form the assembled pattern, was the one that caused the match to occur.

You should realise that large numbers of alternations are processed in perl's regular expression engine in O(n) time, not O(1). If you are still having performance problems, you should look at using a trie. Note that Perl's own regular expression engine will implement trie optimisations in perl 5.10 (they are already available in perl 5.9.3 if you want to try them out). `Regexp::Assemble` will do the right thing when it knows it's running on a trie'd perl. (At least in some version after this one).

Some more examples of usage appear in the accompanying README. If that file is not easy to access locally, you can find it on a web repository such as <http://search.cpan.org/dist/Regexp-Assemble/README> or <http://cpan.uwinnipeg.ca/htdocs/Regexp-Assemble/README.html>.

See also "LIMITATIONS".

## Methods

### add(LIST)

Takes a string, breaks it apart into a set of tokens (respecting meta characters) and inserts the resulting list into the `R::A` object. It uses a naive regular expression to lex the string that may be fooled complex expressions (specifically, it will fail to lex nested parenthetical expressions such as `ab(cd(ef)?gh)ij` correctly). If this is the case, the end of the string will not be tokenised correctly and returned as one long string.

On the one hand, this may indicate that the patterns you are trying to feed the `R::A` object are too complex. Simpler patterns might allow the algorithm to work more effectively and perform more reductions in the resulting pattern.

On the other hand, you can supply your own pattern to perform the lexing if you need. The test suite contains an example of a lexer pattern that will match one level of nested parentheses.

Note that there is an internal optimisation that will bypass a much of the lexing process. If a string contains no `\` (backslash), `[` (open square bracket), `(` (open paren), `?` (question mark), + (plus), `*` (star) or `{` (open curly), a character split will be performed directly.

A list of strings may be supplied, thus you can pass it a file handle of a file opened for reading:

```
$re->add( '\d+-\d+-\d+-\d+\.example\.com' );
$re->add( <IN> );
```

If the file is very large, it may be more efficient to use a `while` loop, to read the file line-by-line:

```
$re->add($_) while <IN>;
```

The `add` method will chomp the lines automatically. If you do not want this to occur (you want to keep the record separator), then disable `chomp`ing.

```
$re->chomp(0);
$re->add($_) while <IN>;
```

This method is chainable.

**add_file(FILENAME [...])**

Takes a list of file names. Each file is opened and read line by line. Each line is added to the assembly.

```
$r->add_file( 'file.1', 'file.2' );
```

If a file cannot be opened, the method will croak. If you cannot afford to let this happen then you should wrap the call in a `eval` block.

Chomping happens automatically unless you the `chomp(0)` method to disable it. By default, input lines are read according to the value of the `input_record_separator` attribute (if defined), and will otherwise fall back to the current setting of the system `$/` variable. The record separator may also be specified on each call to `add_file`. Internally, the routine `local`ises the value of `$/` to whatever is required, for the duration of the call.

An alternate calling mechanism using a hash reference is available. The recognised keys are:

file Reference to a list of file names, or the name of a single file.

```
$r->add_file({file => ['file.1', 'file.2', 'file.3']});
$r->add_file({file => 'file.n'});
```

input_record_separator

If present, indicates what constitutes a line

```
$r->add_file({file => 'data.txt', input_record_separator => ':' });
```

rs An alias for input_record_separator (mnemonic: same as the English variable names).

```
$r->add_file( {
  file => [ 'pattern.txt', 'more.txt' ],
  input_record_separator  => "\r\n",
});
```

*clone()*

Clones the contents of a Regexp::Assemble object and creates a new object (in other words it performs a deep copy).

If the Storable module is installed, its dclone method will be used, otherwise the cloning will be performed using a pure perl approach.

You can use this method to take a snapshot of the patterns that have been added so far to an object, and generate an assembly from the clone. Additional patterns may to be added to the original object afterwards.

```
my $re = $main->clone->re();
$main->add( 'another-pattern-\\d+' );
```

**insert(LIST)**

Takes a list of tokens representing a regular expression and stores them in the object. Note: you should not pass it a bare regular expression, such as `ab+c?d*e`. You must pass it as a list of tokens, *e.g.* `('a', 'b+', 'c?', 'd*', 'e')`.

This method is chainable, *e.g.*:

```
my $ra = Regexp::Assemble->new
  ->insert( qw[ a b+ c? d* e ] )
  ->insert( qw[ a c+ d+ e* f ] );
```

Lexing complex patterns with metacharacters and so on can consume a significant proportion of the overall time to build an assembly. If you have the information available in a tokenised form, calling `insert` directly can be a big win.

**lexstr**

Use the `lexstr` method if you are curious to see how a pattern gets tokenised. It takes a scalar on input, representing a pattern, and returns a reference to an array, containing the tokenised pattern. You can recover the original pattern by performing a `join`:

```
my @token = $re->lexstr($pattern);
my $new_pattern = join( '', @token );
```

If the original pattern contains unnecessary backslashes, or `\x4b` escapes, or quotemeta escapes (`\Q...\E`) the resulting pattern may not be identical.

Call `lexstr` does not add the pattern to the object, it is merely for exploratory purposes. It will, however, update various statistical counters.

**pre_filter(CODE)**

Allows you to install a callback to check that the pattern being loaded contains valid input. It receives the pattern as a whole to be added, before it been tokenised by the lexer. It may to return 0 or `undef` to indicate that the pattern should not be added, any true value indicates that the contents are fine.

A filter to strip out trailing comments (marked by #):

```
$re->pre_filter( sub { $_[0] =~ s/\s*#.*$//; 1 } );
```

A filter to ignore blank lines:

```
$re->pre_filter( sub { length(shift) } );
```

If you want to remove the filter, pass `undef` as a parameter.

```
$ra->pre_filter(undef);
```

This method is chainable.

**filter(CODE)**

Allows you to install a callback to check that the pattern being loaded contains valid input. It receives a list on input, after it has been tokenised by the lexer. It may to return 0 or undef to indicate that the pattern should not be added, any true value indicates that the contents are fine.

If you know that all patterns you expect to assemble contain a restricted set of of tokens (e.g. no spaces), you could do the following:

```
$ra->filter(sub { not grep { / / } @_ });
```

or

```
sub only_spaces_and_digits {
  not grep { ![\d ] } @_
}
$ra->filter( \&only_spaces_and_digits );
```

These two examples will silently ignore faulty patterns, If you want the user to be made aware of the problem you should raise an error (via `warn` or `die`), log an error message, whatever is best. If you want to remove a filter, pass `undef` as a parameter.

```
$ra->filter(undef);
```

This method is chainable.

**as_string**

Assemble the expression and return it as a string. You may want to do this if you are writing the pattern to a file. The following arguments can be passed to control the aspect of the resulting pattern:

**indent**, the number of spaces used to indent nested grouping of a pattern. Use this to produce a pretty-printed pattern (for some definition of ''pretty''). The resulting output is rather verbose. The reason is to

ensure that the metacharacters `(?:` and `)` always occur on otherwise empty lines. This allows you grep the result for an even more synthetic view of the pattern:

```
egrep -v '^ *[()]' <regexp.file>
```

The result of the above is quite readable. Remember to backslash the spaces appearing in your own patterns if you wish to use an indented pattern in an `m/.../x` construct. Indenting is ignored if tracking is enabled.

The **indent** argument takes precedence over the `indent` method/attribute of the object.

Calling this method will drain the internal data structure. Large numbers of patterns can eat a significant amount of memory, and this lets perl recover the memory used for other purposes.

If you want to reduce the pattern *and* continue to add new patterns, clone the object and reduce the clone, leaving the original object intact.

**re**

Assembles the pattern and return it as a compiled RE, using the `qr//` operator.

As with `as_string`, calling this method will reset the internal data structures to free the memory used in assembling the RE.

The **indent** attribute, documented in the `as_string` method, can be used here (it will be ignored if tracking is enabled).

With method chaining, it is possible to produce a RE without having a temporary `Regexp::Assemble` object lying around, *e.g.*:

```
my $re = Regexp::Assemble->new
  ->add( q[ab+cd+e] )
  ->add( q[ac\\d+e] )
  ->add( q[c\\d+e] )
  ->re;
```

The `$re` variable now contains a Regexp object that can be used directly:

```
while( <> ) {
  /$re/ and print "Something in [$_] matched\n";
)
```

The `re` method is called when the object is used in string context (hence, within an `m//` operator), so by and large you do not even need to save the RE in a separate variable. The following will work as expected:

```
my $re = Regexp::Assemble->new->add( qw[ fee fie foe fum ] );
while( <IN> ) {
  if( /($re)/ ) {
    print "Here be giants: $1\n";
  }
}
```

This approach does not work with tracked patterns. The `match` and `matched` methods must be used instead, see below.

**match(SCALAR)**

The following information applies to Perl 5.8 and below. See the section that follows for information on Perl 5.10.

If pattern tracking is in use, you must `use re 'eval'` in order to make things work correctly. At a minimum, this will make your code look like this:

```
my $did_match = do { use re 'eval'; $target =~ /$ra/ }
if( $did_match ) {
    print "matched ", $ra->matched, "\n";
}
```

(The main reason is that the `$^R` variable is currently broken and an ugly workaround that runs some Perl

code during the match is required, in order to simulate what `$^R` should be doing. See Perl bug #32840 for more information if you are curious. The README also contains more information). This bug has been fixed in 5.10.

The important thing to note is that with `use re 'eval'`, THERE ARE SECURITY IMPLICATIONS WHICH YOU IGNORE AT YOUR PERIL. The problem is this: if you do not have strict control over the patterns being fed to `Regexp::Assemble` when tracking is enabled, and someone slips you a pattern such as `/^(?{system 'rm -rf /'})/` and you attempt to match a string against the resulting pattern, you will know Fear and Loathing.

What is more, the `$^R` workaround means that that tracking does not work if you perform a bare `/$re/` pattern match as shown above. You have to instead call the `match` method, in order to supply the necessary context to take care of the tracking housekeeping details.

```
if( defined( my $match = $ra->match($_)) ) {
    print "  $_ matched by $match\n";
}
```

In the case of a successful match, the original matched pattern is returned directly. The matched pattern will also be available through the `matched` method.

(Except that the above is not true for 5.6.0: the `match` method returns true or undef, and the `matched` method always returns undef).

If you are capturing parts of the pattern *e.g.* `foo(bar)rat` you will want to get at the captures. See the `mbegin`, `mend`, `mvar` and `capture` methods. If you are not using captures then you may safely ignore this section.

In 5.10, since the bug concerning `$^R` has been resolved, there is no need to use `re 'eval'` and the assembled pattern does not require any Perl code to be executed during the match.

*new()*

Creates a new `Regexp::Assemble` object. The following optional key/value parameters may be employed. All keys have a corresponding method that can be used to change the behaviour later on. As a general rule, especially if you're just starting out, you don't have to bother with any of these.

**anchor_***, a family of optional attributes that allow anchors (`^`, `\b`, `\Z`...) to be added to the resulting pattern.

**flags**, sets the `imsx` flags to add to the assembled regular expression. Warning: no error checking is done, you should ensure that the flags you pass are understood by the version of Perl you are using. **modifiers** exists as an alias, for users familiar with Regexp::List.

**chomp**, controls whether the pattern should be chomped before being lexed. Handy if you are reading patterns from a file. By default, `chomp`ing is performed (this behaviour changed as of version 0.24, prior versions did not chomp automatically). See also the `file` attribute and the `add_file` method.

**file**, slurp the contents of the specified file and add them to the assembly. Multiple files may be processed by using a list.

```
my $r = Regexp::Assemble->new(file => 're.list');

my $r = Regexp::Assemble->new(file => ['re.1', 're.2']);
```

If you really don't want chomping to occur, you will have to set the `chomp` attribute to 0 (zero). You may also want to look at the `input_record_separator` attribute, as well.

**input_record_separator**, controls what constitutes a record separator when using the `file` attribute or the `add_file` method. May be abbreviated to **rs**. See the `$/` variable in perlvar.

**lookahead**, controls whether the pattern should contain zero-width lookahead assertions (For instance: (?=[abc])(?:bob|alice|charles). This is not activated by default, because in many circumstances the cost of processing the assertion itself outweighs the benefit of its faculty for short-circuiting a match that will fail. This is sensitive to the probability of a match succeeding, so if you're worried about performance you'll

have to benchmark a sample population of targets to see which way the benefits lie.

**track**, controls whether you want know which of the initial patterns was the one that matched. See the `matched` method for more details. Note for version 5.8 of Perl and below, in this mode of operation YOU SHOULD BE AWARE OF THE SECURITY IMPLICATIONS that this entails. Perl 5.10 does not suffer from any such restriction.

**indent**, the number of spaces used to indent nested grouping of a pattern. Use this to produce a pretty-printed pattern. See the `as_string` method for a more detailed explanation.

**pre_filter**, allows you to add a callback to enable sanity checks on the pattern being loaded. This callback is triggered before the pattern is split apart by the lexer. In other words, it operates on the entire pattern. If you are loading patterns from a file, this would be an appropriate place to remove comments.

**filter**, allows you to add a callback to enable sanity checks on the pattern being loaded. This callback is triggered after the pattern has been split apart by the lexer.

**unroll_plus**, controls whether to unroll, for example, `x+` into `x`, `x*`, which may allow additional reductions in the resulting assembled pattern.

**reduce**, controls whether tail reduction occurs or not. If set, patterns like `a(?:bc+d|ec+d)` will be reduced to `a[be]c+d`. That is, the end of the pattern in each part of the b... and d... alternations is identical, and hence is hoisted out of the alternation and placed after it. On by default. Turn it off if you're really pressed for short assembly times.

**lex**, specifies the pattern used to lex the input lines into tokens. You could replace the default pattern by a more sophisticated version that matches arbitrarily nested parentheses, for example.

**debug**, controls whether copious amounts of output is produced during the loading stage or the reducing stage of assembly.

```
my $ra = Regexp::Assemble->new;
my $rb = Regexp::Assemble->new( chomp => 1, debug => 3 );
```

**mutable**, controls whether new patterns can be added to the object after the assembled pattern is generated. DEPRECATED.

This method/attribute will be removed in a future release. It doesn't really serve any purpose, and may be more effectively replaced by cloning an existing `Regexp::Assemble` object and spinning out a pattern from that instead.

*source()*

When using tracked mode, after a successful match is made, returns the original source pattern that caused the match. In Perl 5.10, the `$^R` variable can be used to as an index to fetch the correct pattern from the object.

If no successful match has been performed, or the object is not in tracked mode, this method returns `undef`.

```
my $r = Regexp::Assemble->new->track(1)->add(qw(foo? bar{2} [Rr]at));

for my $w (qw(this food is rather barren)) {
  if ($w =~ /$r/) {
    print "$w matched by ", $r->source($^R), $/;
  }
  else {
    print "$w no match\n";
  }
}
```

*mbegin()*

This method returns a copy of `@-` at the moment of the last match. You should ordinarily not need to bother with this, `mvar` should be able to supply all your needs.

*mend()*

This method returns a copy of `@+` at the moment of the last match.

**mvar(NUMBER)**

The `mvar` method returns the captures of the last match. `mvar(1)` corresponds to `$1`, `mvar(2)` to `$2`, and so on. `mvar(0)` happens to return the target string matched, as a byproduct of walking down the `@-` and `@+` arrays after the match.

If called without a parameter, `mvar` will return a reference to an array containing all captures.

**capture**

The `capture` method returns the the captures of the last match as an array. Unlink `mvar`, this method does not include the matched string. It is equivalent to getting an array back that contains `$1, $2, $3, ...`.

If no captures were found in the match, an empty array is returned, rather than `undef`. You are therefore guaranteed to be able to use `for my $c ($re->capture) { ...` without have to check whether anything was captured.

*matched()*

If pattern tracking has been set, via the `track` attribute, or through the `track` method, this method will return the original pattern of the last successful match. Returns undef match has yet been performed, or tracking has not been enabled.

See below in the NOTES section for additional subtleties of which you should be aware of when tracking patterns.

Note that this method is not available in 5.6.0, due to limitations in the implementation of `(?{...})` at the time.

**Statistics/Reporting routines**

**stats_add**

Returns the number of patterns added to the assembly (whether by `add` or `insert`). Duplicate patterns are not included in this total.

**stats_dup**

Returns the number of duplicate patterns added to the assembly. If non-zero, this may be a sign that something is wrong with your data (or at the least, some needless redundancy). This may occur when you have two patterns (for instance, `a\-b` and `a-b`) which map to the same result.

*stats_raw()*

Returns the raw number of bytes in the patterns added to the assembly. This includes both original and duplicate patterns. For instance, adding the two patterns `ab` and `ab` will count as 4 bytes.

*stats_cooked()*

Return the true number of bytes added to the assembly. This will not include duplicate patterns. Furthermore, it may differ from the raw bytes due to quotemeta treatment. For instance, `abc\,def` will count as 7 (not 8) bytes, because `\,` will be stored as `,`. Also, `\Qa.b\E` is 7 bytes long, however, after the quotemeta directives are processed, `a\.b` will be stored, for a total of 4 bytes.

*stats_length()*

Returns the length of the resulting assembled expression. Until `as_string` or `re` have been called, the length will be 0 (since the assembly will have not yet been performed). The length includes only the pattern, not the additional `(?-xism...)` fluff added by the compilation.

**dup_warn(NUMBER|CODEREF)**

Turns warnings about duplicate patterns on or off. By default, no warnings are emitted. If the method is called with no parameters, or a true parameter, the object will carp about patterns it has already seen. To turn off the warnings, use 0 as a parameter.

```
$r->dup_warn();
```

The method may also be passed a code block. In this case the code will be executed and it will receive a

reference to the object in question, and the lexed pattern.

```
$r->dup_warn(
  sub {
    my $self = shift;
    print $self->stats_add, " patterns added at line $.\n",
        join( '', @_ ), " added previously\n";
  }
)
```

**Anchor routines**

Suppose you wish to assemble a series of patterns that all begin with `^` and end with `$` (anchor pattern to the beginning and end of line). Rather than add the anchors to each and every pattern (and possibly forget to do so when a new entry is added), you may specify the anchors in the object, and they will appear in the resulting pattern, and you no longer need to (or should) put them in your source patterns. For example, the two following snippets will produce identical patterns:

```
$r->add(qw(^this ^that ^them))->as_string;

$r->add(qw(this that them))->anchor_line_begin->as_string;

# both techniques will produce ^th(?:at|em|is)
```

All anchors are possible word (`\b`) boundaries, line boundaries (`^` and `$`) and string boundaries (`\A` and `\Z` (or `\z` if you absolutely need it)).

The shortcut `anchor_`*`mumble`* implies both `anchor_`*`mumble`*`_begin` `anchor_`*`mumble`*`_end` is also available. If different anchors are specified the most specific anchor wins. For instance, if both `anchor_word_begin` and `anchor_line_begin` are specified, `anchor_word_begin` takes precedence.

All the anchor methods are chainable.

**anchor_word_begin**

The resulting pattern will be prefixed with a `\b` word boundary assertion when the value is true. Set to 0 to disable.

```
$r->add('pre')->anchor_word_begin->as_string;
# produces '\bpre'
```

**anchor_word_end**

The resulting pattern will be suffixed with a `\b` word boundary assertion when the value is true. Set to 0 to disable.

```
$r->add(qw(ing tion))
  ->anchor_word_end
  ->as_string; # produces '(?:tion|ing)\b'
```

**anchor_word**

The resulting pattern will be have `\b` word boundary assertions at the beginning and end of the pattern when the value is true. Set to 0 to disable.

```
$r->add(qw(cat carrot)
  ->anchor_word(1)
  ->as_string; # produces '\bca(?:rro)t\b'
```

**anchor_line_begin**

The resulting pattern will be prefixed with a `^` line boundary assertion when the value is true. Set to 0 to disable.

```
$r->anchor_line_begin;
# or
$r->anchor_line_begin(1);
```

**anchor_line_end**

The resulting pattern will be suffixed with a `$` line boundary assertion when the value is true. Set to 0 to disable.

```
# turn it off
$r->anchor_line_end(0);
```

**anchor_line**

The resulting pattern will be have the `^` and `$` line boundary assertions at the beginning and end of the pattern, respectively, when the value is true. Set to 0 to disable.

```
$r->add(qw(cat carrot)
  ->anchor_line
  ->as_string; # produces '^ca(?:rro)t$'
```

**anchor_string_begin**

The resulting pattern will be prefixed with a `\A` string boundary assertion when the value is true. Set to 0 to disable.

```
$r->anchor_string_begin(1);
```

**anchor_string_end**

The resulting pattern will be suffixed with a `\Z` string boundary assertion when the value is true. Set to 0 to disable.

```
# disable the string boundary end anchor
$r->anchor_string_end(0);
```

**anchor_string_end_absolute**

The resulting pattern will be suffixed with a `\z` string boundary assertion when the value is true. Set to 0 to disable.

```
# disable the string boundary absolute end anchor
$r->anchor_string_end_absolute(0);
```

If you don't understand the difference between `\Z` and `\z`, the former will probably do what you want.

**anchor_string**

The resulting pattern will be have the `\A` and `\Z` string boundary assertions at the beginning and end of the pattern, respectively, when the value is true. Set to 0 to disable.

```
$r->add(qw(cat carrot)
  ->anchor_string
  ->as_string; # produces '\Aca(?:rro)t\Z'
```

**anchor_string_absolute**

The resulting pattern will be have the `\A` and `\z` string boundary assertions at the beginning and end of the pattern, respectively, when the value is true. Set to 0 to disable.

```
$r->add(qw(cat carrot)
  ->anchor_string_absolute
  ->as_string; # produces '\Aca(?:rro)t\z'
```

**debug(NUMBER)**

Turns debugging on or off. Statements are printed to the currently selected file handle (STDOUT by default). If you are already using this handle, you will have to arrange to select an output handle to a file of your own choosing, before call the `add`, `as_string` or `re`) functions, otherwise it will scribble all over your carefully formatted output.

- Off. Turns off all debugging output.

- 1

  Add. Trace the addition of patterns.

- 2

  Reduce. Trace the process of reduction and assembly.

- 4

  Lex. Trace the lexing of the input patterns into its constituent tokens.

- 8

  Time. Print to STDOUT the time taken to load all the patterns. This is nothing more than the difference between the time the object was instantiated and the time reduction was initiated.

  ```
  # load=<num>
  ```

  Any lengthy computation performed in the client code will be reflected in this value. Another line will be printed after reduction is complete.

  ```
  # reduce=<num>
  ```

  The above output lines will be changed to `load-epoch` and `reduce-epoch` if the internal state of the object is corrupted and the initial timestamp is lost.

  The code attempts to load Time::HiRes in order to report fractional seconds. If this is not successful, the elapsed time is displayed in whole seconds.

Values can be added (or or'ed together) to trace everything

```
$r->debug(7)->add( '\\d+abc' );
```

Calling `debug` with no arguments turns debugging off.

*dump()*

Produces a synthetic view of the internal data structure. How to interpret the results is left as an exercise to the reader.

```
print $r->dump;
```

**chomp(0|1)**

Turns chomping on or off.

IMPORTANT: As of version 0.24, chomping is now on by default as it makes `add_file` Just Work. The only time you may run into trouble is with `add("\\$/")`. So don't do that, or else explicitly turn off chomping.

To avoid incorporating (spurious) record separators (such as "\n" on Unix) when reading from a file, `add()` chomps its input. If you don't want this to happen, call `chomp` with a false value.

```
$re->chomp(0); # really want the record separators
$re->add(<DATA>);
```

**fold_meta_pairs(NUMBER)**

Determines whether `\s`, `\S` and `\w`, `\W` and `\d`, `\D` are folded into a `.` (dot). Folding happens by default (for reasons of backwards compatibility, even though it is wrong when the `/s` expression modifier is active).

Call this method with a false value to prevent this behaviour (which is only a problem when dealing with `\n` if the `/s` expression modifier is also set).

```
$re->add( '\\w', '\\W' );
my $clone = $re->clone;

$clone->fold_meta_pairs(0);
print $clone->as_string; # prints '.'
```

```
print $re->as_string;     # print '[\W\w]'
```

**indent(NUMBER)**

Sets the level of indent for pretty-printing nested groups within a pattern. See the `as_string` method for more details. When called without a parameter, no indenting is performed.

```
$re->indent( 4 );
print $re->as_string;
```

**lookahead(0|1)**

Turns on zero-width lookahead assertions. This is usually beneficial when you expect that the pattern will usually fail. If you expect that the pattern will usually match you will probably be worse off.

**flags(STRING)**

Sets the flags that govern how the pattern behaves (for versions of Perl up to 5.9 or so, these are `imsx`). By default no flags are enabled.

**modifiers(STRING)**

An alias of the `flags` method, for users familiar with `Regexp::List`.

**track(0|1)**

Turns tracking on or off. When this attribute is enabled, additional housekeeping information is inserted into the assembled expression using `({...})` embedded code constructs. This provides the necessary information to determine which, of the original patterns added, was the one that caused the match.

```
$re->track( 1 );
if( $target =~ /$re/ ) {
  print "$target matched by ", $re->matched, "\n";
}
```

Note that when this functionality is enabled, no reduction is performed and no character classes are generated. In other words, `brag|tag` is not reduced down to `(?:br|t)ag` and `dig|dim` is not reduced to `di[gm]`.

**unroll_plus(0|1)**

Turns the unrolling of plus metacharacters on or off. When a pattern is broken up, `a+` becomes `a`, `a*` (and `b+?` becomes `b`, `b*?`. This may allow the freed `a` to assemble with other patterns. Not enabled by default.

**lex(SCALAR)**

Change the pattern used to break a string apart into tokens. You can examine the `eg/naive` script as a starting point.

**reduce(0|1)**

Turns pattern reduction on or off. A reduced pattern may be considerably shorter than an unreduced pattern. Consider `/sl(?:ip|op|ap)/` *versus* `/sl[aio]p/`. An unreduced pattern will be very similar to those produced by `Regexp::Optimizer`. Reduction is on by default. Turning it off speeds assembly (but assembly is pretty fast — it's the breaking up of the initial patterns in the lexing stage that can consume a non-negligible amount of time).

**mutable(0|1)**

This method has been marked as DEPRECATED. It will be removed in a future release. See the `clone` method for a technique to replace its functionality.

*reset()*

Empties out the patterns that have been `add`ed or `insert`-ed into the object. Does not modify the state of controller attributes such as `debug`, `lex`, `reduce` and the like.

**Default_Lexer**

**Warning:** the `Default_Lexer` function is a class method, not an object method. It is a fatal error to call it as an object method.

The `Default_Lexer` method lets you replace the default pattern used for all subsequently created `Regexp::Assemble` objects. It will not have any effect on existing objects. (It is also possible to

override the lexer pattern used on a per-object basis).

The parameter should be an ordinary scalar, not a compiled pattern. If the pattern fails to match all parts of the string, the missing parts will be returned as single chunks. Therefore the following pattern is legal (albeit rather cork-brained):

```
Regexp::Assemble::Default_Lexer( '\\d' );
```

The above pattern will split up input strings digit by digit, and all non-digit characters as single chunks.

## DIAGNOSTICS

"Cannot pass a C<refname> to Default_Lexer"

You tried to replace the default lexer pattern with an object instead of a scalar. Solution: You probably tried to call `$obj->Default_Lexer`. Call the qualified class method instead `Regexp::Assemble::Default_Lexer`.

"filter method not passed a coderef"

"pre_filter method not passed a coderef"

A reference to a subroutine (anonymous or otherwise) was expected. Solution: read the documentation for the `filter` method.

"duplicate pattern added: /.../"

The `dup_warn` attribute is active, and a duplicate pattern was added (well duh!). Solution: clean your data.

"cannot open [file] for input: [reason]"

The `add_file` method was unable to open the specified file for whatever reason. Solution: make sure the file exists and the script has the required privileges to read it.

## NOTES

This module has been tested successfully with a range of versions of perl, from 5.005_03 to 5.9.3. Use of 5.6.0 is not recommended.

The expressions produced by this module can be used with the PCRE library.

Remember to "double up" your backslashes if the patterns are hard-coded as constants in your program. That is, you should literally `add('a\\d+b')` rather than `add('a\d+b')`. It usually will work either way, but it's good practice to do so.

Where possible, supply the simplest tokens possible. Don't add `X(?-\d+){2}Y` when `X-\d+-\d+Y` will do. The reason is that if you also add `X\d+Z` the resulting assembly changes dramatically: `X(?:(?:-\d+){2}Y|-\d+Z)` *versus* `X-\d+(?:-\d+Y|Z)`. Since R::A doesn't perform enough analysis, it won't "unroll" the `{2}` quantifier, and will fail to notice the divergence after the first `-d\d+`.

Furthermore, when the string 'X-123000P' is matched against the first assembly, the regexp engine will have to backtrack over each alternation (the one that ends in Y **and** the one that ends in Z) before determining that there is no match. No such backtracking occurs in the second pattern: as soon as the engine encounters the 'P' in the target string, neither of the alternations at that point (`-\d+Y` or `Z`) could succeed and so the match fails.

`Regexp::Assemble` does, however, know how to build character classes. Given `a-b`, `axb` and `a\db`, it will assemble these into `a[-\dx]b`. When - (dash) appears as a candidate for a character class it will be the first character in the class. When ^ (circumflex) appears as a candidate for a character class it will be the last character in the class.

It also knows about meta-characters than can "absorb" regular characters. For instance, given `X\d` and `X5`, it knows that `5` can be represented by `\d` and so the assembly is just `X\d`. The "absorbent" meta-characters it deals with are `.`, `\d`, `\s` and `\W` and their complements. It will replace `\d`/`\D`, `\s`/`\S` and `\w`/`\W` by `.` (dot), and it will drop `\d` if `\w` is also present (as will `\D` in the presence of `\W`).

`Regexp::Assemble` deals correctly with `quotemeta`'s propensity to backslash many characters that have no need to be. Backslashes on non-metacharacters will be removed. Similarly, in character classes, a

number of characters lose their magic and so no longer need to be backslashed within a character class. Two common examples are `.` (dot) and `$`. Such characters will lose their backslash.

At the same time, it will also process `\Q...\E` sequences. When such a sequence is encountered, the inner section is extracted and `quotemeta` is applied to the section. The resulting quoted text is then used in place of the original unquoted text, and the `\Q` and `\E` metacharacters are thrown away. Similar processing occurs with the `\U...\E` and `\L...\E` sequences. This may have surprising effects when using a dispatch table. In this case, you will need to know exactly what the module makes of your input. Use the `lexstr` method to find out what's going on:

```
$pattern = join( '', @{$re->lexstr($pattern)} );
```

If all the digits 0..9 appear in a character class, `Regexp::Assemble` will replace them by `\d`. I'd do it for letters as well, but thinking about accented characters and other glyphs hurts my head.

In an alternation, the longest paths are chosen first (for example, `horse|bird|dog`). When two paths have the same length, the path with the most subpaths will appear first. This aims to put the "busiest" paths to the front of the alternation. For example, the list `bad`, `bit`, `few`, `fig` and `fun` will produce the pattern `(?:f(?:ew|ig|un)|b(?:ad|it))`. See *eg/tld* for a real-world example of how alternations are sorted. Once you have looked at that, everything should be crystal clear.

When tracking is in use, no reduction is performed. nor are character classes formed. The reason is that it is too difficult to determine the original pattern afterwards. Consider the two patterns `pale` and `palm`. These should be reduced to `pal[em]`. The final character matches one of two possibilities. To resolve whether it matched an `'e'` or `'m'` would require keeping track of the fact that the pattern finished up in a character class, which would the require a whole lot more work to figure out which character of the class matched. Without character classes it becomes much easier. Instead, `pal(?:e|m)` is produced, which lets us find out more simply where we ended up.

Similarly, `dogfood` and `seafood` should form `(?:dog|sea)food`. When the pattern is being assembled, the tracking decision needs to be made at the end of the grouping, but the tail of the pattern has not yet been visited. Deferring things to make this work correctly is a vast hassle. In this case, the pattern becomes merely `(?:dogfood|seafood`. Tracked patterns will therefore be bulkier than simple patterns.

There is an open bug on this issue:

<http://rt.perl.org/rt3/Ticket/Display.html?id=32840>

If this bug is ever resolved, tracking would become much easier to deal with (none of the `match` hassle would be required – you could just match like a regular RE and it would Just Work).

## SEE ALSO

perlre
: General information about Perl's regular expressions.

re
: Specific information about `use re 'eval'`.

Regex::PreSuf
: `Regex::PreSuf` takes a string and chops it itself into tokens of length 1. Since it can't deal with tokens of more than one character, it can't deal with meta-characters and thus no regular expressions. Which is the main reason why I wrote this module.

Regexp::Optimizer
: `Regexp::Optimizer` produces regular expressions that are similar to those produced by R::A with reductions switched off. It's biggest drawback is that it is exponentially slower than Regexp::Assemble on very large sets of patterns.

Regexp::Parser
: Fine grained analysis of regular expressions.

Regexp::Trie
: Funnily enough, this was my working name for `Regexp::Assemble` during its development. I changed the name because I thought it was too obscure. Anyway, `Regexp::Trie` does much the

same as `Regexp::Optimizer` and `Regexp::Assemble` except that it runs much faster (according to the author). It does not recognise meta characters (that is, 'a+b' is interpreted as 'a\+b').

Text::Trie

`Text::Trie` is well worth investigating. Tries can outperform very bushy (read: many alternations) patterns.

Tree::Trie

`Tree::Trie` is another module that builds tries. The algorithm that `Regexp::Assemble` uses appears to be quite similar to the algorithm described therein, except that `R::A` solves its end-marker problem without having to rewrite the leaves.

## LIMITATIONS

Some mildly complex cases are not handled well. See examples/failure.01.pl and <https://rt.cpan.org/Public/Bug/Display.html?id=104897>.

<Regexp::Assemble> does not attempt to find common substrings. For instance, it will not collapse `/cababab/` down to `/c(?:ab){3}c/`. If there's a module out there that performs this sort of string analysis I'd like to know about it. But keep in mind that the algorithms that do this are very expensive: quadratic or worse.

`Regexp::Assemble` does not interpret meta-character modifiers. For instance, if the following two patterns are given: `X\d` and `X\d+`, it will not determine that `\d` can be matched by `\d+`. Instead, it will produce `X(?:\d|\d+)`. Along a similar line of reasoning, it will not determine that `Z` and `Z\d+` is equivalent to `Z\d*` (It will produce `Z(?:\d+)?` instead).

You cannot remove a pattern that has been added to an object. You'll just have to start over again. Adding a pattern is difficult enough, I'd need a solid argument to convince me to add a `remove` method. If you need to do this you should read the documentation for the `clone` method.

`Regexp::Assemble` does not (yet)? employ the `(?>...)` construct.

The module does not produce POSIX-style regular expressions. This would be quite easy to add, if there was a demand for it.

## BUGS

Patterns that generate look-ahead assertions sometimes produce incorrect patterns in certain obscure corner cases. If you suspect that this is occurring in your pattern, disable lookaheads.

Tracking doesn't really work at all with 5.6.0. It works better in subsequent 5.6 releases. For maximum reliability, the use of a 5.8 release is strongly recommended. Tracking barely works with 5.005_04. Of note, using `\d`–style meta-characters invariably causes panics. Tracking really comes into its own in Perl 5.10.

If you feed `Regexp::Assemble` patterns with nested parentheses, there is a chance that the resulting pattern will be uncompilable due to mismatched parentheses (not enough closing parentheses). This is normal, so long as the default lexer pattern is used. If you want to find out which pattern among a list of 3000 patterns are to blame (speaking from experience here), the *eg/debugging* script offers a strategy for pinpointing the pattern at fault. While you may not be able to use the script directly, the general approach is easy to implement.

The algorithm used to assemble the regular expressions makes extensive use of mutually-recursive functions (that is, A calls B, B calls A, ...) For deeply similar expressions, it may be possible to provoke ''Deep recursion'' warnings.

The module has been tested extensively, and has an extensive test suite (that achieves close to 100% statement coverage), but you never know... A bug may manifest itself in two ways: creating a pattern that cannot be compiled, such as `a\(bc)`, or a pattern that compiles correctly but that either matches things it shouldn't, or doesn't match things it should. It is assumed that Such problems will occur when the reduction algorithm encounters some sort of edge case. A temporary work-around is to disable reductions:

```
my $pattern = $assembler->reduce(0)->re;
```

A discussion about implementation details and where bugs might lurk appears in the README file. If this

file is not available locally, you should be able to find a copy on the Web at your nearest CPAN mirror.

Seriously, though, a number of people have been using this module to create expressions anywhere from 140Kb to 600Kb in size, and it seems to be working according to spec. Thus, I don't think there are any serious bugs remaining.

If you are feeling brave, extensive debugging traces are available to figure out where assembly goes wrong.

Please report all bugs at <http://rt.cpan.org/NoAuth/Bugs.html?Dist=Regexp−Assemble>

Make sure you include the output from the following two commands:

```
perl −MRegexp::Assemble −le 'print $Regexp::Assemble::VERSION'
perl −V
```

There is a mailing list for the discussion of `Regexp::Assemble`. Subscription details are available at <http://listes.mongueurs.net/mailman/listinfo/regexp−assemble>.

## ACKNOWLEDGEMENTS

This module grew out of work I did building access maps for Postfix, a modern SMTP mail transfer agent. See <http://www.postfix.org/> for more information. I used Perl to build large regular expressions for blocking dynamic/residential IP addresses to cut down on spam and viruses. Once I had the code running for this, it was easy to start adding stuff to block really blatant spam subject lines, bogus HELO strings, spammer mailer-ids and more...

I presented the work at the French Perl Workshop in 2004, and the thing most people asked was whether the underlying mechanism for assembling the REs was available as a module. At that time it was nothing more that a twisty maze of scripts, all different. The interest shown indicated that a module was called for. I'd like to thank the people who showed interest. Hey, it's going to make *my* messy scripts smaller, in any case.

Thomas Drugeon was a valuable sounding board for trying out early ideas. Jean Forget and Philippe Blayo looked over an early version. H.Merijn Brandt stopped over in Paris one evening, and discussed things over a few beers.

Nicholas Clark pointed out that while what this module does (?:c|sh)ould be done in perl's core, as per the 2004 TODO, he encouraged me to continue with the development of this module. In any event, this module allows one to gauge the difficulty of undertaking the endeavour in C. I'd rather gouge my eyes out with a blunt pencil.

Paul Johnson settled the question as to whether this module should live in the Regex:: namespace, or Regexp:: namespace. If you're not convinced, try running the following one-liner:

```
perl −le 'print ref qr//'
```

Philippe Bruhat found a couple of corner cases where this module could produce incorrect results. Such feedback is invaluable, and only improves the module's quality.

## Machine-Readable Change Log

The file Changes was converted into Changelog.ini by Module::Metadata::Changes.

## AUTHOR

David Landgren



`http://www.landgren.net/perl/`

If you use this module, I'd love to hear about what you're using it for. If you want to be informed of updates, send me a note.

Ron Savage is co-maint of the module, starting with V 0.36.

## Repository

<https://github.com/ronsavage/Regexp−Assemble.git>

## TODO

1. Tree equivalencies. Currently, /contend/ /content/ /resend/ /resent/ produces (?:conten[dt]|resend[dt]) but it is possible to produce (?:cont|res)en[dt] if one can spot the common tail nodes (and walk back the equivalent paths). Or be by me my => /[bm][ey]/ in the simplest case.

   To do this requires a certain amount of restructuring of the code. Currently, the algorithm uses a two-phase approach. In the first phase, the trie is traversed and reductions are performed. In the second phase, the reduced trie is traversed and the pattern is emitted.

   What has to occur is that the reduction and emission have to occur together. As a node is completed, it is replaced by its string representation. This then allows child nodes to be compared for equality with a simple 'eq'. Since there is only a single traversal, the overall generation time might drop, even though the context baggage required to delve through the tree will be more expensive to carry along (a hash rather than a couple of scalars).

   Actually, a simpler approach is to take on a secret sentinel atom at the end of every pattern, which gives the reduction algorithm sufficient traction to create a perfect trie.

   I'm rewriting the reduction code using this technique.

2. Investigate how (?>foo) works. Can it be applied?

5. How can a tracked pattern be serialised? (Add freeze and thaw methods).

6. Store callbacks per tracked pattern.

12. utf-8... hmmmm...

14. Adding qr//'ed patterns. For example, consider
    `$r->add ( qr/^abc/i )`
    `->add( qr/^abd/ )`
    `->add( qr/^ab e/x );`
    this should admit abc abC aBc aBC abd abe as matches

16. Allow a fast, unsafe tracking mode, that can be used if a(?bc)? can't happen. (Possibly carp if it does appear during traversal)?

17. given a-\d+-\d+-\d+-\d+-b, produce a(?:-\d+){4}-b. Something along the lines of (.{4))(\1+) would let the regexp engine itself be brought to bear on the matter, which is a rather appealing idea. Consider

```
while(/(?!\+)(\S{2,}?)(\1+)/g) { ... $1, $2 ... }
```

    as a starting point.

19. The reduction code has become unbelievably baroque. Adding code to handle (sting,singing,sing) => s(?:(?:ing)?|t)ing was far too difficult. Adding more stuff just breaks existing behaviour. And fixing the ^abcd$ ... bug broke stuff all over again. Now that the corner cases are more clearly identified, a full rewrite of the reduction code is needed. And would admit the possibility of implementing items 1 and 17.

20. Handle debug unrev with a separate bit

23. Japhy's http://www.perlmonks.org/index.pl?node_id=90876 list2range regexp

24. Lookahead assertions contain serious bugs (as shown by assembling powersets. Need to save more context during reduction, which in turn will simplify the preparation of the lookahead classes. See also 19.

26. *_lex()* swamps the overall run-time. It stems from the decision

to use a single regexp to pull apart any pattern. A suite of simpler regexp to pick of parens, char classes, quantifiers and bare tokens may be faster. (This has been implemented as *_fastlex()*, but it's only marginally faster. Perhaps split−by− char and lex a la C?

27. We don't, as yet, unroll_plus a paren e.g. (abc)+?

28. We don't reroll unrolled a a* to a+ in indented or tracked output

29. Use (*MARK n) in blead for tracked patterns, and use (*FAIL) for the unmatchable pattern.

## LICENSE