NAME

Scalar::Validation

Makes validation of scalar values or function (sub) parameters easy, is fast and uses pure Perl.

VERSION

This documentation refers to version 0.601 of Scalar::Validation

SYNOPSIS

```
  use Scalar::Validation qw(:all);

  my $int_1    = validate int_1   => Int   => 123;
  my $float_1  = validate float_1 => Float => 3.1415927;

  my $para_1   = par  parameter_1 => -Range => [1,5] => Int => shift;
  my $exponent = npar -exponent   => -Range => [1,5] => Int => \%options;

  my $para_2     = parameter       parameter_1 => -Range => [1,5] => Int => shift;
  my $exponent_2 = named_parameter -exponent   => -Range => [1,5] => Int => \%options;

  my $int_2    = validate (int_2    => -And => [Scalar => 'Int'],  123);
  my $int_3    = validate (int_3    => -Or  => [Int => 'CodeRef'], 123);
  my $code_ref = validate (code_ref => -Or  => [Int => 'CodeRef'], sub { 123; });

  my $enum_abc = validate (parameter => -Enum => {a => 1, b => 1, c => 1}, 'c');
  my $enum_abc = validate (parameter => -Enum => [ qw (a b c) ], 'c');

  my $int_4    = validate (int_4   => -Optional =>  Int   =>                             undef);
  my $int_5    = validate (int_5   => -Optional => -And   => [Scalar => Int => 0] =>     undef);
  my $int_6    = validate (int_6   => -Optional => -Or    => [Int => CodeRef => 0] =>    undef);
  my $enum_2   = validate (enum_2  => -Optional => -Enum  => {a => 1, b => 1, c => 1} => undef);
  my $range_1  = validate (range_1 => -Optional => -Range => [1,5] => Int =>             undef);
```

Just checks, never dies:

```
  is_valid(valid_1 => Int => 123);   # is valid,     returns 1;
  is_valid(valid_2 => Int => 1.23);  # is not valid, returns 0;
  is_valid(valid_3 => Int => 'a');   # is not valid, returns 0;
  is_valid(valid_4 => Int => undef); # is not valid, returns 0;
```

Free defined rules or wheres only (also for validate(...))

```
  my $value = 2;

  # be careful, doesn't check that $_ is an integer!
  is_valid (free_where_greater_zero => sub { $_ && $_ > 0} => $value);  # is valid, returns 1
```

```
is_valid (free_rule_greater_zero => { -as      => Int =>
                                      -where   => sub { $_ && $_ > 0},
                                      -message => sub { "$_ is not > 0" },
                                    }
          => $value); # is valid, returns 1

my $my_rule = { -as => Int => -where => sub { $_ && $_ > 0} => -message => sub { "$_ is not > 0" };

is_valid (free_rule_greater_zero => $my_rule => $value);              # is valid, returns 1
```

Managing Rules

```
declare_rule (
    NegativeInt => -as      => Int =>           # Parent rule is optional
                   -where   => sub { $_ < 0 },
                   -message => sub { "value '$_' is not a negative integer" },
);

rule_known(Unknown  => 1); # returns 0 (false)
rule_known(Negative => 1); # returns 1 (true)
```

DESCRIPTION

This class implements a fast and flexible validation for scalars. It is implemented functional to get speed and some problems using global rules for all ;).

Validate Subs

Following validation functions exists:

```
validate(...);
  par(...);           # Alias vor validate()
  parameter(...);     # Alias vor validate()

named_parameter(...);
  n_par(...);         # Alias for named_parameter()

is_valid(...);
```

validate(), parameter() and par()

Different names for same functionality. Use like

```
my $var_float = validate ('PI is a float' => Float => $PI);
my $par_int   = par      (par_int         => Int   => shift);
```

First argument is a free name of the check done. If used as parameter check for subs it is the 'name' of the parameter.

Last argument holds the value to be checked. It has to be a scalar, and therefore the module was named `Scalar::Validation`.

Optional last argument: After the value argument can be added a sub to print out an own error message instead of the default error message:

```
my $var_float = validate ('PI is a float' => Float => $PI => sub { 'wrong defined $PI: '.$_ } );
```

All parameters after first before value argument are used to select or define "validation rules".

named_parameter(), n_par()

These subs extract named parameters out of a parameter_hash_ref. Key and value will be deleted from hash during validation. After processing all parameters hash_ref should be empty.

```
my $par_1_int   = npar            (par_1 => Int   => \%parameters);
my $par_2_float = named_parameter (par_2 => Float => \%parameters);
```

First argument ($key) is the key of the parameter. Last argument ($parameters) has to be a hash_ref.

Without these subs you would have to implement for reading par_1:

```
my $key       = 'par_1';
my $value     = delete $parameters->{$key};
my $par_1_int = par ($key => Int   => $value);
```

It could be done in one line, but this line will be complicated and not easy to understand. The key value is needed twiced and that can cause Copy-Paste-Errors.

Dies by error message

On default, application dies with error message, if data checked by `named_parameter(...)` or `validate(...)` is not valid.

```
validate (parameter => -And => [Scalar => 'Int'],  {} );
validate (parameter => -And => [Scalar => 'Int'],  [] );
validate (parameter => -And => [Scalar => 'Int'],  sub { 'abc'; });
```

Just check without die

```
print is_valid(parameter => -And => [Scalar => 'Int'],  123) ." => 123 is int\n";
print is_valid(parameter => -And => [Scalar => 'Int'],  {} ) ." => {} is no scalar\n";
print is_valid(parameter => -And => [Scalar => 'Int'],  [] ) ." => [] is no scalar\n";
print is_valid(parameter => -And => [Scalar => 'Int'],  sub { 'abc'; }) ." => sub { 'abc'; } is no scalar\n";
```

Get validation messages

Message store has to be localized. The only safe way to deal with recursive calls and die! So use a block like this to store messages

```
my @messages;
{
    local ($Scalar::Validation::message_store) = [];

    my $result = is_valid(parameter => -And => [Scalar => 'Int'],  {} );

    @messages = @{validation_messages()} unless $result;
}
```

As parameter check for indexed arguments

It can be also used a parameter check for unnamed and named parameters. `parameters_end \@_;` ensures, that all parameters are processed. Otherwise it rises the usual validation error. Shorthand: `p_end`.

```
sub create_some_polynom {
    my $max_potenz = par maximum_potenz => -Range => [1,5] => Int => shift;
    # additional parameters ...

    p_end \@_;
    # --- run sub -------------------------------------------------

    my $polynom = '';
    map { $polynom .= " + ".int (100*rand())."*x^".($max_potenz-$_); } (0..$max_potenz);

    return $polynom;
};

print create_some_polynom(1)."\n";
print create_some_polynom(2)."\n";
print create_some_polynom(3)."\n";
print create_some_polynom(4)."\n";
print create_some_polynom(5)."\n";
```

Dies by error message

```
print create_some_polynom(5.5)."\n";
print create_some_polynom(6)."\n";
print create_some_polynom(6, 1)."\n";
```

As parameter check for named arguments

Named arguments can also be handled. This needs more runtime than the indexed variant.

`convert_to_named_params()` does a safe conversion by `validate()`.

```
sub create_some_polynom_named {
    my %pars = convert_to_named_params \@_;

    my $max_potenz = npar -maximum_potenz => -Range => [1,5] => Int => \%pars;
    # additional parameters ...

    parameters_end \%pars;

    # --- run sub -------------------------------------------------

    my $polynom = '';
    map { $polynom .= " + ".int (100*rand())."*x^".($max_potenz-$_); } (0..$max_potenz);

    return $polynom;
```

```
  };

  print create_some_polynom_named(-maximum_potenz => 4);
```

Rules

You can and should create your own rules, i.e.

```
  declare_rule (
      Positive =>  -as      => Int =>           # Parent rule is optional
                   -where   => sub { $_ >= 0 },
                   -message => sub { "value $_ is not a positive integer" },
  );

  rule_known(Unknown  => 1); # returns 0 (false)
  rule_known(Positive => 1); # returns 1 (true)
```

The value to be validated is stored in variable `$_`. For `-message =` sub {...}> it is enclosed in single ticks;

Methods for replacing and removing rules will come soon.

Special Rules

There are some special rules, that cannot be changed. Those rules start with an '-' char in front:

```
 -Optional    # value may be undefined. If not, use following rule
 -And         # all rules must be ok
 -Or          # at least one rule must be ok
 -Enum        # for easy defining enumeration on the fly, as array_ref or hash_ref
 -Range       # Intervall: [start, end] => type
 -RefEmpty    # array_ref: scalar (@$array_ref)     == 0
              # hash_ref:  scalar (keys %$hash_ref) == 0
```

Reason is, that they combine other rules or have more or different parameters than a "normal" rule or using own implementation just to speed up.

All normal rules should not start with a '-', but it is not forbidden to do so.

```
  my $var_float = validate ('PI is a float' => -Optional => Float => $PI => sub { 'wrong defined $PI: '.$_ } );
```

This rule does not die, if $PI is undef because of -Optional in front.

Create Own Validation Module

You should not use Scalar::Validation direct in your code.

Better is creating an own module My::Validation, that adds the rules you need and only exports the subs the developers in your project should use:

```
  use My::Validation;

  my $v_my_type = validate v_int => my_type => new MyType();
```

Dealing with XSD

In this case My::Validation creates rules out of XML datatypes after reading in a XSD file. So rules are dynamic and your application can handle different XSD definitions without knowing something about XSD outside of this module.

Also you can filter XSD type contents, i.e. for enmuerations: Allowing not all possible values in UI or remove entries only for compatibility with old versions.

And your Application or GUI doesn't need to know about it.

Validation Modes

There are 4 predefined validation modes:

```
  die
  warn
  ignore
  off
```

Validation Mode 'die' (default)

The validation methods call `croak "validation message";` in case of failures.

More Examples

Have a look into Validation.t to see what else is possible

LICENSE AND COPYRIGHT

DISCLAIMER OF WARRANTY