use v5.36

Marc Espie <espie@openbsd.org>

June 28, 2023

perl

From some time now, perl has got "experimental features", which is a good and a bad thing

good

- Perl is still evolving
- it gets all the good stuff from perl6 over the years
- For instance yada yada

bad

- but they are experimental
- so you can't use them in production
- see switch operator

- Acknowledges "fill in the blanks Rapid Application Developement"
- "Yada Yada" is just . . .

```
1   if ($verbose && ...) {
2           do_something();
3   }
4
5   for my $i (@list) {
6           if ($i =~ m/shouldn't happen/) {
7                   ...;
8           }
9   }
```

That's actually very useful

given/when

There was some kind of "pattern matching" related to keywords `given` and `when` that was available in perl, very similar to what pattern matching looks like in ocaml.

But for various reasons, the experiment didn't pan out! so anyone who's been using these is up shit creek!

semantic versioning

For a long time, use has been used (lol) to indicate semantic variations on perl (like
```
use strict;,
use warnings;
```
or more complicated forms, e.g.,
```
use feature qw(say);
```
The most interesting variation is
```
use v5.something;
```
which does enable various thingies.

fight or flight

For production code, it makes sense to use a recent version and its improvements...
as long as they're not *experimental* !!!

- It's totally different from other languages.
- Prototypes are used to "create syntax" or rather reproduce built-in behaviors.
- Completely esoteric syntax

Example I

```
1   sub mypush (\@@)
2   {
3   }
4   ...
5   mypush @l, 1, 2, 3;
6
```

Example II

```
1   sub try(&@)
2   {
3           my ($try, $catch) = @_;
4           eval { &$try };
5           if ($@) {
6                   &$catch;
7           }
8   }
9
10  sub catch(&)
11  {
12          return $_[0];
13  }
14
15  try {
16          ...
17  } catch {
18          ...
19  }
```

Signatures

- so what's called "prototypes" in other languages is called "signatures" in perl !
- it's ambiguous wrt prototypes
- so accordingly prototypes require annotation e.g.,

```perl
sub try :prototype(&@)($try, $catch)
{
        eval { &$try() };
        if ($@) {
                &$catch;
        }
        ...
}

sub catch :prototype(&)($code)
{
        return $code;
}

```

Converting to 5.36: other stuff

- Every prototype needs to be made unambiguous so, sub foo(...) → sub foo
  :prototype(...).
- Code calls may need explicit parentheses: stuff like

```
1   &$code;
```
  does call code *in the same context* as the parent with the same parameters. Use

```
1   &$code();
```
  instead.
- Object calls through indirect syntax has been deprecated.

Sometimes useful

```perl
1  package OpenBSD::PackingElement::Cwd;
2  sub find_extractible    # forwarder
3  {
4          &OpenBSD::PackingElement::Meta::find_extractible;
5  }
```

- Again linked to built-ins.
- Printing to a file looks like

```
1  print $fh "result is ", $i, "\n";
```

- as opposed to the less fancy

```
1  $fh->print("result is ", $i, "\n");
```

Modelled after C++

- named scalar parameters like `$x`
- unnamed parameters if unused
- default values for parameters with `$x = value`

But perl!

Can slurp renaming parameters with either

```
1  sub f($x, $y, @l)
```

or

```
1  sub f($x, $y, %h)
```

What about @_

```
1  Implicit use of @_ in subroutine entry with signatured subroutine
2      is experimental at a line 5.
```

Why use signatures I

It makes perl code looks almost normal! before

```perl
 1    sub set
 2    {
 3            my ($self, $set) = @_;
 4            delete $self->{object};
 5            $self->{set} = $set;
 6            return $self;
 7    }
 8
 9    sub object
10    {
11            my ($self, $object) = @_;
12            delete $self->{set};
13            $self->{object} = $object;
14            return $self;
15    }
16
17    sub what
18    {
19            my ($self, $what) = @_;
20            $self->{what} = $what;
21            return $self;
22    }
23
24    sub new
25    {
26            my $class = shift;
27
```

```
28          bless {}, $class;
29     }
30
```

after

```perl
 1    sub set($self, $set)
 2    {
 3            delete $self->{object};
 4            $self->{set} = $set;
 5            return $self;
 6    }
 7
 8    sub object($self, $object)
 9    {
10            delete $self->{set};
11            $self->{object} = $object;
12            return $self;
13    }
14
15    sub what($self, $what = undef)
16    {
17            $self->{what} = $what;
18            return $self;
19    }
20
21    sub new($class)
22    {
23            bless {}, $class;
24    }
25
```

- because of OO
- sometimes a base method does nothing

```perl
1  package OpenBSD::PackingElement;
2  # $self->find_dependencies($state, $l, $checker, $pkgname)
3  sub find_dependencies($, $, $, $, $)
4  {
5  }
6
```

- Errors happen at runtime, so difficult to catch them all

```
1  Too few arguments for subroutine 'main::f'
2      (got 2; expected 3) at b line 11.
```

- Much harder on lambdas

```
1  Too few arguments for subroutine 'main::__ANON__'
2      (got 2; expected 3) at b line 11.
3
```

(it would be great to annotate the name of the anonymous routine with line number and filename)

Paradigm shift

- perl has long been fuzzy on parameter numbers
- some default interfaces use this to "tack on" parameters (e.g. signals)
- some OO code has been designed to take advantage of this

- like in C, normal signal handlers get the signal number (but it's not guaranteed)
- There are extra signal handlers for `__DIE__` or `__WARN__` and these take optional messages.

- in the worst case, you can slurp stuff with @
- but it's better to track the error and get the right number of parameters
- sometimes variations, because of time (a parameter is no longer used)
- or variation in parameters used for a constructor subclass

```perl
1  package OpenBSD::ProgressMeter;
2  sub new($class, $state)
3  {
4          # now saves state
5  }
6
7  sub for_list($self, $msg, $l, $code) # + $state
```

We can use @ to get parameters through until we have a default value

```perl
1
2  package OpenBSD::PackingList;
3
4  sub read($a, $u, $code = \&defaultCode)
5
6  sub fromfile($a, $fname, $code = \&defaultCode)
7
8  package OpenBSD::PackageLocation;
9  sub grabPlist($self, $code = \&OpenBSD::PackingList::defaultCode)
10
11 sub grabPlist($self, @code)
```

Grepping

```
1   package OpenBSD::PackingElement::Cwd;
2   sub find_extractible    # forwarder
3   {
4           &OpenBSD::PackingElement::Meta::find_extractible;
5   }

1
2           my $handler = sub {          # SIGHANDLER
3                   $state->{received} = shift;
4                   $state->errsay("Interrupted");
5                   if ($state->{hardkill}) {
6                           delete $state->{hardkill};
7                           return;
8                   }
9                   $state->{interrupted}++;
10          };

11
12          local $SIG{'INT'} = $handler;
13          local $SIG{'QUIT'} = $handler;
14          local $SIG{'HUP'} = $handler;
```

- insufficient coverage of the code base
- should look at `Devel::Cover` and subclasses
- static validation is lagging (`perlcritic`).
- future optimizations.