NAME

Data::OptList - parse and validate simple name/value option pairs

VERSION

version 0.109

SYNOPSIS

```
use Data::OptList;

my $options = Data::OptList::mkopt([
qw(key1 key2 key3 key4),
key5 => { ... },
key6 => [ ... ],
key7 => sub { ... },
key8 => { ... },
key8 => [ ... ],
]);
```

...is the same thing, more or less, as:

```
my $options = [
[ key1 => undef, ],
[ key2 => undef, ],
[ key3 => undef, ],
[ key4 => undef, ],
[ key5 => { ... }, ],
[ key6 => [ ... ], ],
[ key7 => sub { ... }, ],
[ key8 => { ... }, ],
[ key8 => [ ... ], ],
]);
```

DESCRIPTION

Hashes are great for storing named data, but if you want more than one entry for a name, you have to use a list of pairs. Even then, this is really boring to write:

```
$values = [
foo => undef,
bar => undef,
baz => undef,
xyz => { ... },
];
```

Just look at all those undefs! Don't worry, we can get rid of those:

```
$values = [
map { $_ => undef } qw(foo bar baz),
xyz => { ... },
];
```

Aaaauuugh! We've saved a little typing, but now it requires thought to read, and thinking is even worse than typing... and it's got a bug! It looked right, didn't it? Well, the `xyz => { ... }` gets consumed by the map, and we don't get the data we wanted.

With Data::OptList, you can do this instead:

```
$values = Data::OptList::mkopt([
qw(foo bar baz),
xyz => { ... },
]);
```

This works by assuming that any defined scalar is a name and any reference following a name is

its value.

FUNCTIONS

mkopt

```
my $opt_list = Data::OptList::mkopt($input, \%arg);
```

Valid arguments are:

```
moniker - a word used in errors to describe the opt list; encouraged
require_unique - if true, no name may appear more than once
must_be - types to which opt list values are limited (described below)
name_test - a coderef used to test whether a value can be a name
(described below, but you probably don't want this)
```

This produces an array of arrays; the inner arrays are name/value pairs. Values will be either "undef" or a reference.

Positional parameters may be used for compatibility with the old `mkopt` interface:

```
my $opt_list = Data::OptList::mkopt($input, $moniker, $req_uni, $must_be);
```

Valid values for `$input`:

```
undef -> []
hashref -> [ [ key1 => value1 ] ... ] # non-ref values become undef
arrayref -> every name followed by a non-name becomes a pair: [ name => ref ]
every name followed by undef becomes a pair: [ name => undef ]
otherwise, it becomes [ name => undef ] like so:
[ "a", "b", [ 1, 2 ] ] -> [ [ a => undef ], [ b => [ 1, 2 ] ] ]
```

By default, a *name* is any defined non-reference. The `name_test` parameter can be a code ref that tests whether the argument passed it is a name or not. This should be used rarely. Interactions between `require_unique` and `name_test` are not yet particularly elegant, as `require_unique` just tests string equality. **This may change.**

The `must_be` parameter is either a scalar or array of scalars; it defines what kind(s) of refs may be values. If an invalid value is found, an exception is thrown. If no value is passed for this argument, any reference is valid. If`must_be` specifies that v alues must be CODE, HASH, ARRAY, or SCALAR, then Params::Util is used to check whether the given value can provide that interface. Otherwise, it checks that the given value is an object of the kind.

In other words:

```
[ qw(SCALAR HASH Object::Known) ]
```

Means:

```
_SCALAR0($value) or _HASH($value) or _INSTANCE($value, 'Object::Known')
```

mkopt_hash

```
my $opt_hash = Data::OptList::mkopt_hash($input, $moniker, $must_be);
```

Given valid `"mkopt"` input, this routine returns a reference to a hash. It will throw an exception if any name has more than one value.

EXPORTS

Both `mkopt` and `mkopt_hash` may be exported on request.

AUTHOR

Ricardo Signes <rjbs@cpan.org>

COPYRIGHT AND LICENSE