NAME

overload - Package for overloading Perl operations

SYNOPSIS

```
    package SomeThing;

    use overload
 '+' => \&myadd,
 '-' => \&mysub;
 # etc
    ...

    package main;
    $a = new SomeThing 57;
    $b=5+$a;
    ...
    if (overload::Overloaded $b) {...}
    ...
    $strval = overload::StrVal $b;
```

DESCRIPTION

Declaration of overloaded functions

The compilation directive

```
    package Number;
    use overload
 "+" => \&add,
 "*=" => "muas";
```

declares function Number::add() for addition, and method muas() in the "class" `Number` (or one of its base classes) for the assignment form *= of multiplication.

Arguments of this directive come in (key, value) pairs. Legal values are values legal inside a `&{ ... }` call, so the name of a subroutine, a reference to a subroutine, or an anonymous subroutine will all work. Note that values specified as strings are interpreted as methods, not subroutines. Legal keys are listed below.

The subroutine `add` will be called to execute $a+$b if $a is a reference to an object blessed into the package `Number`, or if $a is not an object from a package with defined mathemagic addition, but $b is a reference to a `Number`. It can also be called in other situations, like `$a+=7`, or `$a++`. See *MAGIC AUTOGENERATION*. (Mathemagical methods refer to methods triggered by an overloaded mathematical operator.)

Since overloading respects inheritance via the @ISA hierarchy, the above declaration would also trigger overloading of + and *= in all the packages which inherit from `Number`.

Calling Conventions for Binary Operations

The functions specified in the `use overload ...` directive are called with three (in one particular case with four, see *Last Resort*) arguments. If the corresponding operation is binary, then the first two arguments are the two arguments of the operation. However, due to general object calling conventions, the first argument should always be an object in the package, so in the situation of `7+$a`, the order of the arguments is interchanged. It probably does not matter when implementing the addition method, but whether the arguments are reversed is vital to the subtraction method. The method can query this information by examining the third argument, which can take three different values:

FALSE

the order of arguments is as in the current operation.

TRUE

the arguments are reversed.

`undef`

the current operation is an assignment variant (as in $a+=7), but the usual function is called instead. This additional information can be used to generate some optimizations. Compare *Calling Conventions for Mutators*.

Calling Conventions for Unary Operations

Unary operation are considered binary operations with the second argument being `undef`. Thus the functions that overloads `{"++"}` is called with arguments `($a,undef,'')` when $a++ is executed.

Calling Conventions for Mutators

Two types of mutators have different calling conventions:

++ and --

The routines which implement these operators are expected to actually *mutate* their arguments. So, assuming that $obj is a reference to a number,

```
sub incr { my $n = $ {$_[0]}; ++$n; $_[0] = bless \$n}
```

is an appropriate implementation of overloaded ++. Note that

```
sub incr { ++$ {$_[0]} ; shift }
```

is OK if used with preincrement and with postincrement. (In the case of postincrement a copying will be performed, see *Copy Constructor*.)

x= and other assignment versions

There is nothing special about these methods. They may change the value of their arguments, and may leave it as is. The result is going to be assigned to the value in the left-hand-side if different from this value.

This allows for the same method to be used as overloaded += and +. Note that this is *allowed*, but not recommended, since by the semantic of *Fallback* Perl will call the method for + anyway, if += is not overloaded.

Warning. Due to the presence of assignment versions of operations, routines which may be called in assignment context may create self-referential structures. Currently Perl will not free self-referential structures until cycles are `explicitly` broken. You may get problems when traversing your structures too.

Say,

```
use overload '+' => sub { bless [ \$_[0], \$_[1] ] };
```

is asking for trouble, since for code `$obj += $foo` the subroutine is called as `$obj = add($obj, $foo, undef)`, or `$obj = [\$obj, \$foo]`. If using such a subroutine is an important optimization, one can overload += explicitly by a non-"optimized" version, or switch to non-optimized version if `not defined $_[2]` (see *Calling Conventions for Binary Operations*).

Even if no *explicit* assignment-variants of operators are present in the script, they may be generated by the optimizer. Say, `",$obj,"` or `',' . $obj . ','` may be both optimized to

```
my $tmp = ',' . $obj;    $tmp .= ',';
```

Overloadable Operations

The following symbols can be specified in `use overload` directive:

* *Arithmetic operations*

```
"+", "+=", "-", "-=", "*", "*=", "/", "/=", "%", "%=",
"**", "**=", "<<", "<<=", ">>", ">>=", "x", "x=", ".", ".=",
```

For these operations a substituted non-assignment variant can be called if the assignment variant is not available. Methods for operations `+`, `-`, `+=`, and `-=` can be called to automatically generate increment and decrement methods. The operation `-` can be used to autogenerate missing methods for unary minus or `abs`.

See *MAGIC AUTOGENERATION*, *Calling Conventions for Mutators* and *Calling Conventions for Binary Operations*) for details of these substitutions.

* *Comparison operations*

```
"<",  "<=", ">",  ">=", "==", "!=", "<=>",
"lt", "le", "gt", "ge", "eq", "ne", "cmp",
```

If the corresponding "spaceship" variant is available, it can be used to substitute for the missing operation. During `sort`ing arrays, `cmp` is used to compare values subject to `use overload`.

* *Bit operations*

```
"&", "&=", "^", "^=", "|", "|=", "neg", "!", "~",
```

`neg` stands for unary minus. If the method for `neg` is not specified, it can be autogenerated using the method for subtraction. If the method for `!` is not specified, it can be autogenerated using the methods for `bool`, or `""`, or `0+`.

The same remarks in *Arithmetic operations* about assignment-variants and autogeneration apply for bit operations `"&"`, `"^"`, and `"|"` as well.

* *Increment and decrement*

```
"++", "--",
```

If undefined, addition and subtraction methods can be used instead. These operations are called both in prefix and postfix form.

* *Transcendental functions*

```
"atan2", "cos", "sin", "exp", "abs", "log", "sqrt", "int"
```

If `abs` is unavailable, it can be autogenerated using methods for "<" or "<=>" combined with either unary minus or subtraction.

Note that traditionally the Perl function *int* rounds to 0, thus for floating-point-like types one should follow the same semantic. If `int` is unavailable, it can be autogenerated using the overloading of `0+`.

* *Boolean, string and numeric conversion*

```
'bool', '""', '0+',
```

If one or two of these operations are not overloaded, the remaining ones can be used instead. `bool` is used in the flow control operators (like `while`) and for the ternary `?:` operation. These functions can return any arbitrary Perl value. If the corresponding operation for this value is overloaded too, that operation will be called again with this value.

As a special case if the overload returns the object itself then it will be used directly. An

overloaded conversion returning the object is probably a bug, because you're likely to get something that looks like `YourPackage=HASH(0x8172b34)`.

* *Iteration*

```
"<>"
```

If not overloaded, the argument will be converted to a filehandle or glob (which may require a stringification). The same overloading happens both for the *read-filehandle* syntax `<$var>` and *globbing* syntax `<${var}>`.

BUGS Even in list context, the iterator is currently called only once and with scalar context.

* *Dereferencing*

```
'${}', '@{}', '%{}', '&{}', '*{}'.
```

If not overloaded, the argument will be dereferenced *as is*, thus should be of correct type. These functions should return a reference of correct type, or another object with overloaded dereferencing.

As a special case if the overload returns the object itself then it will be used directly (provided it is the correct type).

The dereference operators must be specified explicitly they will not be passed to "nomethod".

* *Special*

```
"nomethod", "fallback", "=",
```

see *SPECIAL SYMBOLS FOR `use overload`*.

See *Fallback* for an explanation of when a missing method can be autogenerated.

A computer-readable form of the above table is available in the hash %overload::ops, with values being space-separated lists of names:

```
with_assign   => '+ - * / % ** << >> x .',
assign    => '+= -= *= /= %= **= <<= >>= x= .=',
num_comparison   => '< <= > >= == !=',
'3way_comparison'=> '<=> cmp',
str_comparison   => 'lt le gt ge eq ne',
binary    => '& &= | |= ^ ^=',
unary    => 'neg ! ~',
mutators   => '++ --',
func    => 'atan2 cos sin exp abs log sqrt',
conversion   => 'bool "" 0+',
iterators   => '<>',
dereferencing   => '${} @{} %{} &{} *{}',
special   => 'nomethod fallback ='
```

Inheritance and overloading

Inheritance interacts with overloading in two ways.

Strings as values of `use overload` directive

If `value` in

```
use overload key => value;
```

is a string, it is interpreted as a method name.

Overloading of an operation is inherited by derived classes

Any class derived from an overloaded class is also overloaded. The set of overloaded methods is the union of overloaded methods of all the ancestors. If some method is overloaded in several ancestor, then which description will be used is decided by the usual inheritance rules:

If A inherits from B and C (in this order), B overloads + with `\&D::plus_sub`, and C overloads + by `"plus_meth"`, then the subroutine `D::plus_sub` will be called to implement operation + for an object in package A.

Note that since the value of the `fallback` key is not a subroutine, its inheritance is not governed by the above rules. In the current implementation, the value of `fallback` in the first overloaded ancestor is used, but this is accidental and subject to change.

SPECIAL SYMBOLS FOR use overload

Three keys are recognized by Perl that are not covered by the above description.

Last Resort

`"nomethod"` should be followed by a reference to a function of four parameters. If defined, it is called when the overloading mechanism cannot find a method for some operation. The first three arguments of this function coincide with the arguments for the corresponding method if it were found, the fourth argument is the symbol corresponding to the missing method. If several methods are tried, the last one is used. Say, `1-$a` can be equivalent to

```
&nomethodMethod($a,1,1,"-")
```

if the pair `"nomethod" => "nomethodMethod"` was specified in the `use overload` directive.

The `"nomethod"` mechanism is *not* used for the dereference operators (${} @{} %{} &{} *{}).

If some operation cannot be resolved, and there is no function assigned to `"nomethod"`, then an exception will be raised via die()-- unless `"fallback"` was specified as a key in `use overload` directive.

Fallback

The key `"fallback"` governs what to do if a method for a particular operation is not found. Three different cases are possible depending on the value of `"fallback"`:

* `undef`

  Perl tries to use a substituted method (see *MAGIC AUTOGENERATION*). If this fails, it then tries to calls `"nomethod"` value; if missing, an exception will be raised.

* TRUE

  The same as for the `undef` value, but no exception is raised. Instead, it silently reverts to what it would have done were there no `use overload` present.

* defined, but FALSE

  No autogeneration is tried. Perl tries to call `"nomethod"` value, and if this is missing, raises an exception.

Note. `"fallback"` inheritance via @ISA is not carved in stone yet, see *Inheritance and overloading*.

Copy Constructor

The value for `"="` is a reference to a function with three arguments, i.e., it looks like the other values in `use overload`. However, it does not overload the Perl assignment operator. This would go against Camel hair.

This operation is called in the situations when a mutator is applied to a reference that shares its object with some other reference, such as

```
$a=$b;
++$a;
```

To make this change $a and not change $b, a copy of $$a is made, and $a is assigned a reference to this new object. This operation is done during execution of the ++$a, and not during the assignment, (so before the increment $$a coincides with $$b). This is only done if ++ is expressed via a method for `'++'` or `'+='` (or `nomethod`). Note that if this operation is expressed via `'+'` a nonmutator, i.e., as in

```
$a=$b;
$a=$a+1;
```

then $a does not reference a new copy of $$a, since $$a does not appear as lvalue when the above code is executed.

If the copy constructor is required during the execution of some mutator, but a method for `'='` was not specified, it can be autogenerated as a string copy if the object is a plain scalar or a simple assignment if it is not.

Example

The actually executed code for

```
$a=$b;
       Something else which does not modify $a or $b....
++$a;
```

may be

```
$a=$b;
       Something else which does not modify $a or $b....
$a = $a->clone(undef,"");
       $a->incr(undef,"");
```

if $b was mathemagical, and `'++'` was overloaded with `\&incr`, `'='` was overloaded with `\&clone`.

Same behaviour is triggered by `$b = $a++`, which is consider a synonym for `$b = $a; ++$a`.

MAGIC AUTOGENERATION

If a method for an operation is not found, and the value for `"fallback"` is TRUE or undefined, Perl tries to autogenerate a substitute method for the missing operation based on the defined operations. Autogenerated method substitutions are possible for the following operations:

Assignment forms of arithmetic operations

$a+=$b can use the method for `"+"` if the method for `"+="` is not defined.

Conversion operations

String, numeric, and boolean conversion are calculated in terms of one another if not all of them are defined.

Increment and decrement

The ++$a operation can be expressed in terms of `$a+=1` or `$a+1`, and `$a--` in terms of `$a-=1` and `$a-1`.

`abs($a)`

can be expressed in terms of `$a<0` and `-$a` (or `0-$a`).

Unary minus

can be expressed in terms of subtraction.

Negation

`!` and `not` can be expressed in terms of boolean conversion, or string or numerical conversion.

Concatenation

can be expressed in terms of string conversion.

Comparison operations

can be expressed in terms of its "spaceship" counterpart: either <=> or `cmp`:

```
<, >, <=, >=, ==, !=  in terms of <=>
lt, gt, le, ge, eq, ne  in terms of cmp
```

Iterator

```
<>    in terms of builtin operations
```

Dereferencing

```
${} @{} %{} &{} *{}  in terms of builtin
operations
```

Copy operator

can be expressed in terms of an assignment to the dereferenced value, if this value is a scalar and not a reference, or simply a reference assignment otherwise.

Minimal set of overloaded operations

Since some operations can be automatically generated from others, there is a minimal set of operations that need to be overloaded in order to have the complete set of overloaded operations at one's disposal. Of course, the autogenerated operations may not do exactly what the user expects. See *MAGIC AUTOGENERATION* above. The minimal set is:

```
+ - * / % ** << >> x
<=> cmp
& | ^ ~
atan2 cos sin exp log sqrt int
```

Additionally, you need to define at least one of string, boolean or numeric conversions because any one can be used to emulate the others. The string conversion can also be used to emulate concatenation.

Losing overloading

The restriction for the comparison operation is that even if, for example, `cmp' should return a blessed reference, the autogenerated `lt' function will produce only a standard logical value based on the numerical value of the result of `cmp'. In particular, a working numeric conversion is needed in this case (possibly expressed in terms of other conversions).

Similarly, `.=` and `x=` operators lose their mathemagical properties if the string conversion substitution is applied.

When you chop() a mathemagical object it is promoted to a string and its mathemagical properties are lost. The same can happen with other operations as well.

Run-time Overloading

Since all `use` directives are executed at compile-time, the only way to change overloading during run-time is to

```
eval 'use overload "+" => \&addmethod';
```

You can also use

```
eval 'no overload "+", "--", "<="';
```

though the use of these constructs during run-time is questionable.

Public functions

Package `overload.pm` provides the following public functions:

overload::StrVal(arg)

Gives string value of `arg` as in absence of stringify overloading. If you are using this to get the address of a reference (useful for checking if two references point to the same thing) then you may be better off using `Scalar::Util::refaddr()`, which is faster.

overload::Overloaded(arg)

Returns true if `arg` is subject to overloading of some operations.

overload::Method(obj,op)

Returns `undef` or a reference to the method that implements `op`.

Overloading constants

For some applications, the Perl parser mangles constants too much. It is possible to hook into this process via `overload::constant()` and `overload::remove_constant()` functions.

These functions take a hash as an argument. The recognized keys of this hash are:

integer

to overload integer constants,

float

to overload floating point constants,

binary

to overload octal and hexadecimal constants,

q

to overload q-quoted strings, constant pieces of `qq`- and `qx`-quoted strings and here-documents,

qr

to overload constant pieces of regular expressions.

The corresponding values are references to functions which take three arguments: the first one is the *initial* string form of the constant, the second one is how Perl interprets this constant, the third one is how the constant is used. Note that the initial string form does not contain string delimiters, and has backslashes in backslash-delimiter combinations stripped (thus the value of delimiter is not relevant for processing of this string). The return value of this function is how this constant is going to be interpreted by Perl. The third argument is undefined unless for overloaded `q`- and `qr`- constants, it is

q in single-quote context (comes from strings, regular expressions, and single-quote HERE documents), it is `tr` for arguments of `tr`/y operators, it is `s` for right-hand side of `s`-operator, and it is `qq` otherwise.

Since an expression `"ab$cd,,"` is just a shortcut for `'ab' . $cd . ',,'`, it is expected that overloaded constant strings are equipped with reasonable overloaded catenation operator, otherwise absurd results will result. Similarly, negative numbers are considered as negations of positive constants.

Note that it is probably meaningless to call the functions overload::constant() and overload::remove_constant() from anywhere but import() and unimport() methods. From these methods they may be called as

```
sub import {
  shift;
  return unless @_;
  die "unknown import: @_" unless @_ == 1 and $_[0] eq ':constant';
  overload::constant integer => sub {Math::BigInt->new(shift)};
}
```

BUGS Currently overloaded-ness of constants does not propagate into `eval '...'`.

IMPLEMENTATION

What follows is subject to change RSN.

The table of methods for all operations is cached in magic for the symbol table hash for the package. The cache is invalidated during processing of `use overload`, `no overload`, new function definitions, and changes in @ISA. However, this invalidation remains unprocessed until the next `bless`ing into the package. Hence if you want to change overloading structure dynamically, you'll need an additional (fake) `bless`ing to update the table.

(Every SVish thing has a magic queue, and magic is an entry in that queue. This is how a single variable may participate in multiple forms of magic simultaneously. For instance, environment variables regularly have two forms at once: their %ENV magic and their taint magic. However, the magic which implements overloading is applied to the stashes, which are rarely used directly, thus should not slow down Perl.)

If an object belongs to a package using overload, it carries a special flag. Thus the only speed penalty during arithmetic operations without overloading is the checking of this flag.

In fact, if `use overload` is not present, there is almost no overhead for overloadable operations, so most programs should not suffer measurable performance penalties. A considerable effort was made to minimize the overhead when overload is used in some package, but the arguments in question do not belong to packages using overload. When in doubt, test your speed with `use overload` and without it. So far there have been no reports of substantial speed degradation if Perl is compiled with optimization turned on.

There is no size penalty for data if overload is not used. The only size penalty if overload is used in some package is that *all* the packages acquire a magic during the next `bless`ing into the package. This magic is three-words-long for packages without overloading, and carries the cache table if the package is overloaded.

Copying ($a=$b) is shallow; however, a one-level-deep copying is carried out before any operation that can imply an assignment to the object $a (or $b) refers to, like $a++. You can override this behavior by defining your own copy constructor (see *Copy Constructor*).

It is expected that arguments to methods that are not explicitly supposed to be changed are constant (but this is not enforced).

Metaphor clash

One may wonder why the semantic of overloaded = is so counter intuitive. If it *looks* counter intuitive to you, you are subject to a metaphor clash.

Here is a Perl object metaphor:

object is a reference to blessed data

and an arithmetic metaphor:

object is a thing by itself.

The *main* problem of overloading = is the fact that these metaphors imply different actions on the assignment `$a = $b` if $a and $b are objects. Perl-think implies that $a becomes a reference to whatever $b was referencing. Arithmetic-think implies that the value of "object" $a is changed to become the value of the object $b, preserving the fact that $a and $b are separate entities.

The difference is not relevant in the absence of mutators. After a Perl-way assignment an operation which mutates the data referenced by $a would change the data referenced by $b too. Effectively, after `$a = $b` values of $a and $b become *indistinguishable*.

On the other hand, anyone who has used algebraic notation knows the expressive power of the arithmetic metaphor. Overloading works hard to enable this metaphor while preserving the Perlian way as far as possible. Since it is not possible to freely mix two contradicting metaphors, overloading allows the arithmetic way to write things *as far as all the mutators are called via overloaded access only*. The way it is done is described in *Copy Constructor*.

If some mutator methods are directly applied to the overloaded values, one may need to *explicitly unlink* other values which references the same value:

```
$a = new Data 23;
...
$b = $a;  # $b is "linked" to $a
...
$a = $a->clone; # Unlink $b from $a
$a->increment_by(4);
```

Note that overloaded access makes this transparent:

```
$a = new Data 23;
$b = $a;  # $b is "linked" to $a
$a += 4;  # would unlink $b automagically
```

However, it would not make

```
$a = new Data 23;
$a = 4;  # Now $a is a plain 4, not 'Data'
```

preserve "objectness" of $a. But Perl *has* a way to make assignments to an object do whatever you want. It is just not the overload, but tie()ing interface (see *"tie" in perlfunc*). Adding a FETCH() method which returns the object itself, and STORE() method which changes the value of the object, one can reproduce the arithmetic metaphor in its completeness, at least for variables which were tie()d from the start.

(Note that a workaround for a bug may be needed, see *BUGS*.)

Cookbook

Please add examples to what follows!

Two-face scalars

Put this in *two_face.pm* in your Perl library directory:

```
package two_face;  # Scalars with separate string and
                               # numeric values.
sub new { my $p = shift; bless [@_], $p }
use overload '""' => \&str, '0+' => \&num, fallback => 1;
sub num {shift->[1]}
sub str {shift->[0]}
```

Use it as follows:

```
require two_face;
my $seven = new two_face ("vii", 7);
printf "seven=$seven, seven=%d, eight=%d\n", $seven, $seven+1;
print "seven contains `i'\n" if $seven =~ /i/;
```

(The second line creates a scalar which has both a string value, and a numeric value.) This prints:

```
seven=vii, seven=7, eight=8
seven contains `i'
```

Two-face references

Suppose you want to create an object which is accessible as both an array reference and a hash reference, similar to the *pseudo-hash* builtin Perl type. Let's make it better than a pseudo-hash by allowing index 0 to be treated as a normal element.

```
package two_refs;
use overload '%{}' => \&gethash, '@{}' => sub { $ {shift()} };
sub new {
  my $p = shift;
  bless \ [@_], $p;
}
sub gethash {
  my %h;
  my $self = shift;
  tie %h, ref $self, $self;
  \%h;
}

sub TIEHASH { my $p = shift; bless \ shift, $p }
my %fields;
my $i = 0;
$fields{$_} = $i++ foreach qw{zero one two three};
sub STORE {
  my $self = ${shift()};
  my $key = $fields{shift()};
  defined $key or die "Out of band access";
  $$self->[$key] = shift;
}
sub FETCH {
  my $self = ${shift()};
  my $key = $fields{shift()};
  defined $key or die "Out of band access";
  $$self->[$key];
```

```
}
```

Now one can access an object using both the array and hash syntax:

```
my $bar = new two_refs 3,4,5,6;
$bar->[2] = 11;
$bar->{two} == 11 or die 'bad hash fetch';
```

Note several important features of this example. First of all, the *actual* type of $bar is a scalar reference, and we do not overload the scalar dereference. Thus we can get the *actual* non-overloaded contents of $bar by just using `$$bar` (what we do in functions which overload dereference). Similarly, the object returned by the TIEHASH() method is a scalar reference.

Second, we create a new tied hash each time the hash syntax is used. This allows us not to worry about a possibility of a reference loop, which would lead to a memory leak.

Both these problems can be cured. Say, if we want to overload hash dereference on a reference to an object which is *implemented* as a hash itself, the only problem one has to circumvent is how to access this *actual* hash (as opposed to the *virtual* hash exhibited by the overloaded dereference operator). Here is one possible fetching routine:

```
sub access_hash {
  my ($self, $key) = (shift, shift);
  my $class = ref $self;
  bless $self, 'overload::dummy'; # Disable overloading of %{}
  my $out = $self->{$key};
  bless $self, $class; # Restore overloading
  $out;
}
```

To remove creation of the tied hash on each access, one may an extra level of indirection which allows a non-circular structure of references:

```
package two_refs1;
use overload '%{}' => sub { ${shift()}->[1] },
             '@{}' => sub { ${shift()}->[0] };
sub new {
  my $p = shift;
  my $a = [@_];
  my %h;
  tie %h, $p, $a;
  bless \ [$a, \%h], $p;
}
sub gethash {
  my %h;
  my $self = shift;
  tie %h, ref $self, $self;
  \%h;
}

sub TIEHASH { my $p = shift; bless \ shift, $p }
my %fields;
my $i = 0;
$fields{$_} = $i++ foreach qw{zero one two three};
sub STORE {
  my $a = ${shift()};
```

```
  my $key = $fields{shift()};
  defined $key or die "Out of band access";
  $a->[$key] = shift;
}
sub FETCH {
  my $a = ${shift()};
  my $key = $fields{shift()};
  defined $key or die "Out of band access";
  $a->[$key];
}
```

Now if $baz is overloaded like this, then `$baz` is a reference to a reference to the intermediate array, which keeps a reference to an actual array, and the access hash. The tie()ing object for the access hash is a reference to a reference to the actual array, so

- There are no loops of references.
- Both "objects" which are blessed into the class `two_refs1` are references to a reference to an array, thus references to a *scalar*. Thus the accessor expression `$$foo->[$ind]` involves no overloaded operations.

Symbolic calculator

Put this in *symbolic.pm* in your Perl library directory:

```
package symbolic;  # Primitive symbolic calculator
use overload nomethod => \&wrap;

sub new { shift; bless ['n', @_] }
sub wrap {
  my ($obj, $other, $inv, $meth) = @_;
  ($obj, $other) = ($other, $obj) if $inv;
  bless [$meth, $obj, $other];
}
```

This module is very unusual as overloaded modules go: it does not provide any usual overloaded operators, instead it provides the *Last Resort* operator `nomethod`. In this example the corresponding subroutine returns an object which encapsulates operations done over the objects: `new symbolic 3` contains `['n', 3]`, `2 + new symbolic 3` contains `['+', 2, ['n', 3]]`.

Here is an example of the script which "calculates" the side of circumscribed octagon using the above package:

```
require symbolic;
my $iter = 1;   # 2**($iter+2) = 8
my $side = new symbolic 1;
my $cnt = $iter;

while ($cnt--) {
  $side = (sqrt(1 + $side**2) - 1)/$side;
}
print "OK\n";
```

The value of $side is

```
['/', ['-', ['sqrt', ['+', 1, ['**', ['n', 1], 2]],
             undef], 1], ['n', 1]]
```

Note that while we obtained this value using a nice little script, there is no simple way to *use* this value. In fact this value may be inspected in debugger (see *perldebug*), but only if `bareStringify` **O** ption is set, and not via p command.

If one attempts to print this value, then the overloaded operator `""` will be called, which will call `nomethod` operator. The result of this operator will be stringified again, but this result is again of type `symbolic`, which will lead to an infinite loop.

Add a pretty-printer method to the module *symbolic.pm*:

```
sub pretty {
  my ($meth, $a, $b) = @{+shift};
  $a = 'u' unless defined $a;
  $b = 'u' unless defined $b;
  $a = $a->pretty if ref $a;
  $b = $b->pretty if ref $b;
  "[$meth $a $b]";
}
```

Now one can finish the script by

```
print "side = ", $side->pretty, "\n";
```

The method `pretty` is doing object-to-string conversion, so it is natural to overload the operator `""` using this method. However, inside such a method it is not necessary to pretty-print the *components* $a and $b of an object. In the above subroutine `"[$meth $a $b]"` is a catenation of some strings and components $a and $b. If these components use overloading, the catenation operator will look for an overloaded operator `.`; if not present, it will look for an overloaded operator `""`. Thus it is enough to use

```
use overload nomethod => \&wrap, '""' => \&str;
sub str {
  my ($meth, $a, $b) = @{+shift};
  $a = 'u' unless defined $a;
  $b = 'u' unless defined $b;
  "[$meth $a $b]";
}
```

Now one can change the last line of the script to

```
print "side = $side\n";
```

which outputs

```
side = [/ [- [sqrt [+ 1 [** [n 1 u] 2]] u] 1] [n 1 u]]
```

and one can inspect the value in debugger using all the possible methods.

Something is still amiss: consider the loop variable $cnt of the script. It was a number, not an object. We cannot make this value of type `symbolic`, since then the loop will not terminate.

Indeed, to terminate the cycle, the $cnt should become false. However, the operator `bool` for checking falsity is overloaded (this time via overloaded `""`), and returns a long string, thus any object of type `symbolic` is true. To overcome this, we need a way to compare an object to 0. In fact, it is easier to write a numeric conversion routine.

Here is the text of *symbolic.pm* with such a routine added (and slightly modified str()):

```
package symbolic;  # Primitive symbolic calculator
use overload
  nomethod => \&wrap, '""' => \&str, '0+' => \&num;

sub new { shift; bless ['n', @_] }
sub wrap {
  my ($obj, $other, $inv, $meth) = @_;
  ($obj, $other) = ($other, $obj) if $inv;
  bless [$meth, $obj, $other];
}
sub str {
  my ($meth, $a, $b) = @{+shift};
  $a = 'u' unless defined $a;
  if (defined $b) {
    "[$meth $a $b]";
  } else {
    "[$meth $a]";
  }
}
my %subr = ( n => sub {$_[0]},
      sqrt => sub {sqrt $_[0]},
      '-' => sub {shift() - shift()},
      '+' => sub {shift() + shift()},
      '/' => sub {shift() / shift()},
      '*' => sub {shift() * shift()},
      '**' => sub {shift() ** shift()},
    );
sub num {
  my ($meth, $a, $b) = @{+shift};
  my $subr = $subr{$meth}
    or die "Do not know how to ($meth) in symbolic";
  $a = $a->num if ref $a eq __PACKAGE__;
  $b = $b->num if ref $b eq __PACKAGE__;
  $subr->($a,$b);
}
```

All the work of numeric conversion is done in %subr and num(). Of course, %subr is not complete, it contains only operators used in the example below. Here is the extra-credit question: why do we need an explicit recursion in num()? (Answer is at the end of this section.)

Use this module like this:

```
require symbolic;
my $iter = new symbolic 2; # 16-gon
my $side = new symbolic 1;
my $cnt = $iter;

while ($cnt) {
  $cnt = $cnt - 1;  # Mutator `--' not implemented
  $side = (sqrt(1 + $side**2) - 1)/$side;
}
printf "%s=%f\n", $side, $side;
printf "pi=%f\n", $side*(2**($iter+2));
```

It prints (without so many line breaks)

```
[/ [- [sqrt [+ 1 [** [/ [- [sqrt [+ 1 [** [n 1] 2]]] 1]
   [n 1]] 2]]] 1]
   [/ [- [sqrt [+ 1 [** [n 1] 2]]] 1] [n 1]]]=0.198912
pi=3.182598
```

The above module is very primitive. It does not implement mutator methods (++, -= and so on), does not do deep copying (not required without mutators!), and implements only those arithmetic operations which are used in the example.

To implement most arithmetic operations is easy; one should just use the tables of operations, and change the code which fills %subr to

```
my %subr = ( 'n' => sub {$_[0]} );
foreach my $op (split " ", $overload::ops{with_assign}) {
  $subr{$op} = $subr{"$op="} = eval "sub {shift() $op shift()}";
}
my @bins = qw(binary 3way_comparison num_comparison str_comparison);
foreach my $op (split " ", "@overload::ops{ @bins }") {
  $subr{$op} = eval "sub {shift() $op shift()}";
}
foreach my $op (split " ", "@overload::ops{qw(unary func)}") {
  print "defining `$op'\n";
  $subr{$op} = eval "sub {$op shift()}";
}
```

Due to *Calling Conventions for Mutators*, we do not need anything special to make += and friends work, except filling += entry of %subr, and defining a copy constructor (needed since Perl has no way to know that the implementation of `'+='` does not mutate the argument, compare *Copy Constructor*).

To implement a copy constructor, add `'=' => \&cpy` to `use overload` line, and code (this code assumes that mutators change things one level deep only, so recursive copying is not needed):

```
sub cpy {
  my $self = shift;
  bless [@$self], ref $self;
}
```

To make ++ and `--` work, we need to implement actual mutators, either directly, or in `nomethod`. We continue to do things inside `nomethod`, thus add

```
  if ($meth eq '++' or $meth eq '--') {
    @$obj = ($meth, (bless [@$obj]), 1); # Avoid circular reference
    return $obj;
  }
```

after the first line of wrap(). This is not a most effective implementation, one may consider

```
sub inc { $_[0] = bless ['++', shift, 1]; }
```

instead.

As a final remark, note that one can fill %subr by

```
my %subr = ( 'n' => sub {$_[0]} );
foreach my $op (split " ", $overload::ops{with_assign}) {
  $subr{$op} = $subr{"$op="} = eval "sub {shift() $op shift()}";
}
```

```
my @bins = qw(binary 3way_comparison num_comparison str_comparison);
foreach my $op (split " ", "@overload::ops{ @bins }") {
  $subr{$op} = eval "sub {shift() $op shift()}";
}
foreach my $op (split " ", "@overload::ops{qw(unary func)}") {
  $subr{$op} = eval "sub {$op shift()}";
}
$subr{'++'} = $subr{'+'};
$subr{'--'} = $subr{'-'};
```

This finishes implementation of a primitive symbolic calculator in 50 lines of Perl code. Since the numeric values of subexpressions are not cached, the calculator is very slow.

Here is the answer for the exercise: In the case of str(), we need no explicit recursion since the overloaded `.`-operator will fall back to an existing overloaded operator `""`. Overloaded arithmetic operators *do not* fall back to numeric conversion if `fallback` is not explicitly requested. Thus without an explicit recursion num() would convert `['+', $a, $b]` to `$a + $b`, which would just rebuild the argument of num().

If you wonder why defaults for conversion are different for str() and num(), note how easy it was to write the symbolic calculator. This simplicity is due to an appropriate choice of defaults. One extra note: due to the explicit recursion num() is more fragile than sym(): we need to explicitly check for the type of $a and $b. If components $a and $b happen to be of some related type, this may lead to problems.

Really symbolic calculator

One may wonder why we call the above calculator symbolic. The reason is that the actual calculation of the value of expression is postponed until the value is *used*.

To see it in action, add a method

```
sub STORE {
  my $obj = shift;
  $#$obj = 1;
  @$obj->[0,1] = ('=', shift);
}
```

to the package `symbolic`. After this change one can do

```
my $a = new symbolic 3;
my $b = new symbolic 4;
my $c = sqrt($a**2 + $b**2);
```

and the numeric value of $c becomes 5. However, after calling

```
$a->STORE(12);  $b->STORE(5);
```

the numeric value of $c becomes 13. There is no doubt now that the module symbolic provides a *symbolic* calculator indeed.

To hide the rough edges under the hood, provide a tie()d interface to the package `symbolic` (compare with *Metaphor clash*). Add methods

```
sub TIESCALAR { my $pack = shift; $pack->new(@_) }
sub FETCH { shift }
sub nop {  }  # Around a bug
```

(the bug is described in *BUGS*). One can use this new interface as

```
tie $a, 'symbolic', 3;
tie $b, 'symbolic', 4;
$a->nop;  $b->nop; # Around a bug

my $c = sqrt($a**2 + $b**2);
```

Now numeric value of $c is 5. After `$a = 12; $b = 5` the numeric value of $c becomes 13. To insulate the user of the module add a method

```
sub vars { my $p = shift; tie($_, $p), $_->nop foreach @_; }
```

Now

```
my ($a, $b);
symbolic->vars($a, $b);
my $c = sqrt($a**2 + $b**2);

$a = 3; $b = 4;
printf "c5  %s=%f\n", $c, $c;

$a = 12; $b = 5;
printf "c13  %s=%f\n", $c, $c;
```

shows that the numeric value of $c follows changes to the values of $a and $b.

AUTHOR

Ilya Zakharevich <*ilya@math.mps.ohio-state.edu*>.

DIAGNOSTICS

When Perl is run with the **-Do** switch or its equivalent, overloading induces diagnostic messages.

Using the `m` command of Perl debugger (see *perldebug*) one can deduce which operations are overloaded (and which ancestor triggers this overloading). Say, if `eq` is overloaded, then the method `(eq` is shown by debugger. The method `()` corresponds to the `fallback` key (in fact a presence of this method shows that this package has overloading enabled, and it is what is used by the `Overloaded` function of module `overload`).

The module might issue the following warnings:

Odd number of arguments for overload::constant

(W) The call to overload::constant contained an odd number of arguments. The arguments should come in pairs.

`%s' is not an overloadable type

(W) You tried to overload a constant type the overload package is unaware of.

`%s' is not a code reference

(W) The second (fourth, sixth, ...) argument of overload::constant needs to be a code reference. Either an anonymous subroutine, or a reference to a subroutine.

BUGS

Because it is used for overloading, the per-package hash %OVERLOAD now has a special meaning in Perl. The symbol table is filled with names looking like line-noise.

For the purpose of inheritance every overloaded package behaves as if `fallback` is present (possibly undefined). This may create interesting effects if some package is not overloaded, but inherits from two overloaded packages.

Relation between overloading and tie()ing is broken. Overloading is triggered or not basing on the *previous* class of tie()d value.

This happens because the presence of overloading is checked too early, before any tie()d access is attempted. If the FETCH()ed class of the tie()d value does not change, a simple workaround is to access the value immediately after tie()ing, so that after this call the *previous* class coincides with the current one.

Needed: a way to fix this without a speed penalty.

Barewords are not covered by overloaded string constants.

This document is confusing. There are grammos and misleading language used in places. It would seem a total rewrite is needed.