NAME

bigint - Transparent BigInteger support for Perl

SYNOPSIS

```
use bigint;

$x = 2 + 4.5,"\n";                    # BigInt 6
print 2 ** 512,"\n";                  # really is what you think it is
print inf + 42,"\n";                  # inf
print NaN * 7,"\n";                   # NaN
print hex("0x1234567890123490"),"\n"; # Perl v5.10.0 or later

{
  no bigint;
  print 2 ** 256,"\n";                # a normal Perl scalar now
}

# Import into current package:
use bigint qw/hex oct/;
print hex("0x1234567890123490"),"\n";
print oct("01234567890123490"),"\n";
```

DESCRIPTION

All operators (including basic math operations) except the range operator `..` are overloaded. Integer constants are created as proper BigInts.

Floating point constants are truncated to integer. All parts and results of expressions are also truncated.

Unlike *integer*, this pragma creates integer constants that are only limited in their size by the available memory and CPU time.

use integer vs. use bigint

There is one small difference between `use integer` and `use bigint`: the former will not affect assignments to variables and the return value of some functions. `bigint` truncates these results to integer too:

```
# perl -Minteger -wle 'print 3.2'
3.2
# perl -Minteger -wle 'print 3.2 + 0'
3
# perl -Mbigint -wle 'print 3.2'
3
# perl -Mbigint -wle 'print 3.2 + 0'
3

# perl -Mbigint -wle 'print exp(1) + 0'
2
# perl -Mbigint -wle 'print exp(1)'
2
# perl -Minteger -wle 'print exp(1)'
2.71828182845905
# perl -Minteger -wle 'print exp(1) + 0'
2
```

In practice this makes seldom a difference as **parts and results** of expressions will be truncated anyway, but this can, for instance, affect the return value of subroutines:

```
sub three_integer { use integer; return 3.2; }
sub three_bigint { use bigint; return 3.2; }

print three_integer(), " ", three_bigint(),"\n";    # prints "3.2 3"
```

Options

bigint recognizes some options that can be passed while loading it via use. The options can (currently) be either a single letter form, or the long form. The following options exist:

a or accuracy

This sets the accuracy for all math operations. The argument must be greater than or equal to zero. See Math::BigInt's bround() function for details.

```
perl -Mbigint=a,2 -le 'print 12345+1'
```

Note that setting precision and accuracy at the same time is not possible.

p or precision

This sets the precision for all math operations. The argument can be any integer. Negative values mean a fixed number of digits after the dot, and are <B>ignored</B> since all operations happen in integer space. A positive value rounds to this digit left from the dot. 0 or 1 mean round to integer and are ignore like negative values.

See Math::BigInt's bfround() function for details.

```
perl -Mbignum=p,5 -le 'print 123456789+123'
```

Note that setting precision and accuracy at the same time is not possible.

t or trace

This enables a trace mode and is primarily for debugging bigint or Math::BigInt.

hex

Override the built-in hex() method with a version that can handle big integers. This overrides it by exporting it to the current package. Under Perl v5.10.0 and higher, this is not so necessary, as hex() is lexically overridden in the current scope whenever the bigint pragma is active.

oct

Override the built-in oct() method with a version that can handle big integers. This overrides it by exporting it to the current package. Under Perl v5.10.0 and higher, this is not so necessary, as oct() is lexically overridden in the current scope whenever the bigint pragma is active.

l, lib, try or only

Load a different math lib, see *Math Library*.

```
perl -Mbigint=lib,GMP -e 'print 2 ** 512'
perl -Mbigint=try,GMP -e 'print 2 ** 512'
perl -Mbigint=only,GMP -e 'print 2 ** 512'
```

Currently there is no way to specify more than one library on the command line. This means the following does not work:

```
perl -Mbignum=l,GMP,Pari -e 'print 2 ** 512'
```

This will be hopefully fixed soon ;)

v or version

This prints out the name and version of all modules used and then exits.

```
perl -Mbigint=v
```

Math Library

Math with the numbers is done (by default) by a module called Math::BigInt::Calc. This is equivalent to saying:

```
use bigint lib => 'Calc';
```

You can change this by using:

```
use bignum lib => 'GMP';
```

The following would first try to find Math::BigInt::Foo, then Math::BigInt::Bar, and when this also fails, revert to Math::BigInt::Calc:

```
use bigint lib => 'Foo,Math::BigInt::Bar';
```

Using `lib` warns if none of the specified libraries can be found and *Math::BigInt* did fall back to one of the default libraries. To suppress this warning, use `try` instead:

```
use bignum try => 'GMP';
```

If you want the code to die instead of falling back, use `only` instead:

```
use bignum only => 'GMP';
```

Please see respective module documentation for further details.

Internal Format

The numbers are stored as objects, and their internals might change at anytime, especially between math operations. The objects also might belong to different classes, like Math::BigInt, or Math::BigInt::Lite. Mixing them together, even with normal scalars is not extraordinary, but normal and expected.

You should not depend on the internal format, all accesses must go through accessor methods. E.g. looking at $x->{sign} is not a good idea since there is no guaranty that the object in question has such a hash key, nor is a hash underneath at all.

Sign

The sign is either '+', '-', 'NaN', '+inf' or '-inf'. You can access it with the sign() method.

A sign of 'NaN' is used to represent the result when input arguments are not numbers or as a result of 0/0. '+inf' and '-inf' represent plus respectively minus infinity. You will get '+inf' when dividing a positive number by 0, and '-inf' when dividing any negative number by 0.

Method calls

Since all numbers are now objects, you can use all functions that are part of the BigInt API. You can only use the bxxx() notation, and not the fxxx() notation, though.

But a warning is in order. When using the following to make a copy of a number, only a shallow copy will be made.

```
$x = 9; $y = $x;
$x = $y = 7;
```

Using the copy or the original with overloaded math is okay, e.g. the following work:

```
$x = 9; $y = $x;
print $x + 1, " ", $y,"\n";     # prints 10 9
```

but calling any method that modifies the number directly will result in **both** the original and the copy being destroyed:

```
$x = 9; $y = $x;
print $x->badd(1), " ", $y,"\n";        # prints 10 10

$x = 9; $y = $x;
print $x->binc(1), " ", $y,"\n";        # prints 10 10

$x = 9; $y = $x;
print $x->bmul(2), " ", $y,"\n";        # prints 18 18
```

Using methods that do not modify, but test that the contents works:

```
$x = 9; $y = $x;
$z = 9 if $x->is_zero();                # works fine
```

See the documentation about the copy constructor and = in overload, as well as the documentation in BigInt for further details.

Methods

inf()

A shortcut to return Math::BigInt->binf(). Useful because Perl does not always handle bareword `inf` properly.

NaN()

A shortcut to return Math::BigInt->bnan(). Useful because Perl does not always handle bareword `NaN` properly.

e

```
# perl -Mbigint=e -wle 'print e'
```

Returns Euler's number e, aka exp(1). Note that under bigint, this is truncated to an integer, and hence simple '2'.

PI

```
# perl -Mbigint=PI -wle 'print PI'
```

Returns PI. Note that under bigint, this is truncated to an integer, and hence simple '3'.

bexp()

```
bexp($power,$accuracy);
```

Returns Euler's number e raised to the appropriate power, to the wanted accuracy.

Note that under bigint, the result is truncated to an integer.

Example:

```
# perl -Mbigint=bexp -wle 'print bexp(1,80)'
```

bpi()

```
bpi($accuracy);
```

Returns PI to the wanted accuracy. Note that under bigint, this is truncated to an integer, and hence simple '3'.

Example:

```
# perl -Mbigint=bpi -wle 'print bpi(80)'
```

upgrade()

Return the class that numbers are upgraded to, is in fact returning `$Math::BigInt::upgrade`.

in_effect()

```
use bigint;

print "in effect\n" if bigint::in_effect;       # true
{
  no bigint;
  print "in effect\n" if bigint::in_effect;     # false
}
```

Returns true or false if `bigint` is in effect in the current scope.

This method only works on Perl v5.9.4 or later.

CAVEATS

Operator vs literal overloading

`bigint` works by overloading handling of integer and floating point literals, converting them to *Math::BigInt* objects.

This means that arithmetic involving only string values or string literals will be performed using Perl's built-in operators.

For example:

```
use bignum;
my $x = "900000000000000009";
my $y = "900000000000000007";
print $x - $y;
```

will output `0` on default 32-bit builds, since `bigint` never sees the string literals. To ensure the expression is all treated as `Math::BigInt` objects, use a literal number in the expression:

```
print +(0+$x) - $y;
```

ranges

Perl does not allow overloading of ranges, so you can neither safely use ranges with bigint endpoints, nor is the iterator variable a bigint.

```
use 5.010;
for my $i (12..13) {
  for my $j (20..21) {
    say $i ** $j;  # produces a floating-point number,
                   # not a big integer
  }
}
```

in_effect()

This method only works on Perl v5.9.4 or later.

hex()/oct()

`bigint` overrides these routines with versions that can also handle big integer values. Under Perl prior to version v5.9.4, however, this will not happen unless you specifically ask for it with the two import tags "hex" and "oct" - and then it will be global and cannot be disabled inside a scope with "no bigint":

```
use bigint qw/hex oct/;

print hex("0x1234567890123456");
{
        no bigint;
        print hex("0x1234567890123456");
}
```

The second call to hex() will warn about a non-portable constant.

Compare this to:

```
use bigint;

# will warn only under Perl older than v5.9.4
print hex("0x1234567890123456");
```

MODULES USED

`bigint` is just a thin wrapper around various modules of the Math::BigInt family. Think of it as the head of the family, who runs the shop, and orders the others to do the work.

The following modules are currently used by bigint:

```
Math::BigInt::Lite      (for speed, and only if it is loadable)
Math::BigInt
```

EXAMPLES

Some cool command line examples to impress the Python crowd ;) You might want to compare them to the results under -Mbignum or -Mbigrat:

```
perl -Mbigint -le 'print sqrt(33)'
perl -Mbigint -le 'print 2*255'
perl -Mbigint -le 'print 4.5+2*255'
perl -Mbigint -le 'print 3/7 + 5/7 + 8/3'
perl -Mbigint -le 'print 123->is_odd()'
perl -Mbigint -le 'print log(2)'
perl -Mbigint -le 'print 2 ** 0.5'
perl -Mbigint=a,65 -le 'print 2 ** 0.2'
perl -Mbignum=a,65,l,GMP -le 'print 7 ** 7777'
```

LICENSE

This program is free software; you may redistribute it and/or modify it under the same terms as Perl itself.

SEE ALSO

Especially *bigrat* as in `perl -Mbigrat -le 'print 1/3+1/4'` and *bignum* as in `perl -Mbignum -le 'print sqrt(2)'`.

Math::BigInt, *Math::BigRat* and *Math::Big* as well as *Math::BigInt::Pari* and *Math::BigInt::GMP*.

AUTHORS

(C) by Tels *http://bloodgate.com/* in early 2002 - 2007.