Nemo: A computer algebra package for Julia

William B. Hart

September 25, 2015

Contents

1 Introduction

Nemo is a computer algebra package for the Julia programming language (http://julialang.org/).

Nemo is based on the C libraries:

1. FLINT (Fast Library for Number Theory) – http://flintlib.org/
2. ANTIC (Algebraic Number Theory in Code) – http://github.com/wbhart/antic/
3. Arb – http://fredrikj.net/arb/
4. Pari – http://pari.math.u-bordeaux.fr/
5. Singular – http://www.singular.uni-kl.de/

The scope of Nemo is computer algebra in the limited sense of the term, and number theory.

We hope that Nemo will eventually also provide access to the Gap library – http://www.gap-system.org/.

2 The Julia programming language

Over two hundred programming languages were examined before settling on Julia as our choice for the Nemo project.

Some statically compiled languages such as C++, D, Rust and Nimrod could have been reasonable solutions up to a point. Languages such as Scala, Aldor and various others also have strong points.

But Julia is the only language which met all our design criteria.

1. Interactive REPL
2. Jit compiled
3. Fast native C/C++ interop
4. Dependent type system
5. Familiar imperative syntax
6. Open source implementation
7. Regular releases
8. Decent sized community
9. Portable to OSX, Windows, Linux, BSD

Julia started life in 2009 with a first public release in 2012. It's main architects are Jeff Bezanson, Stefan Karpinski and Viral B. Shah, though there are well over 150 contributors to date.

Julia is based on the lightning fast LLVM compiler back end and just-in-time compiler. It is capable of producing code which runs as fast as native C code.

Julia sports hundreds of contributed libraries, a package manager and even a Jupyter based graphical interface.

Hundreds of thousands of lines of code have already been written in Julia. The language is already used in production environments, including big data and parallel environments. The Julia website gets over 150,000 visits a month. There are Julia users groups worldwide and courses taught in Julia in at least Canada and the US.

Most importantly, Julia is designed by mathematicians for mathematicians.

Some of the key features of Julia are briefly discussed below.

2.0.1 Jit compilation

Julia uses just-in-time (JIT) compilation, which allows it to achieve C-like performance in real time without a separate compilation cycle. Jit compilation is achieved through use of the LLVM compiler back end.

LLVM is the 15 year old compiler infrastructure which underlies the Clang C/C++ compiler and components of over a dozen other programming languages.

LLVM's chief architect is Chris Lattner, an Apple employee since 2005, now working on Apple's Swift programming language, which uses LLVM heavily. LLVM itself is reported to be used by Apple for its IOS tool development.

Jit compilation is ideal for mathematical code, since we generally do not care if there is a very short delay for incremental compilation after we hit our Enter key. We care only about how fast our very large loops will take to execute, after jit compilation has happened.

Julia combines LLVM's Jit with dynamic type inference. This is a strategy which allows one to combine most of the benefits of a statically typed language (such as C/C++) with the flexibility and ease of use of dynamically typed languages (such as Python, Javascript, PHP, Perl and Ruby).

To the greatest extent possible, Julia tries to infer the types for each function. However, it allows code to be written without specifying types, if one should so wish. Thus, from the user's perspective it is dynamically typed. But in many cases, exactly the same machine code results as would come from a C compiler and hand-written C.

2.0.2 Interactive console

Julia offers an interactive console, similar to Python. It can also be used from a Jupyter derivative known as IJulia. This can be used on the web or even natively on Windows!

Julia has a console which provides Readline like functionality and offers an online help system.

Most importantly, Julia offers Jit compiled code at the console! You don't have to precompile Julia statically to get C-like performance. What you type at the console executes at lightning speed, immediately.

The console offers exception handling and recovery, pretty printing of types and values, stack traces, profiling and timing, the ability to disassemble the Jit compiled code, and exposes Julia's type inference machinery, introspection capabilities, interactive loading of modules and libraries (including native code) and all of the Julia language, including sophisticated metaprogramming features.

2.0.3 Foreign function interface

Julia allows direct access to C/C++ libraries. There is also a simple mapping from C types to Julia types, and thus complex and costly wrappers are not needed.

Any native dynamic library (such as flint) can be directly accessed from within Julia, so long as it is in the current library path.

There is no additional overhead in accessing native C libraries from within Julia than there would be from a native C program using that library.

2.0.4 Dependent and parametric types

Few languages in the world sport a decent dependent and parametric type system. If they do, it is usually at static compile time only.

The prime example of parametric types for a computer algebraist is when creating a type for polynomials over a ring R. In Julia, we can provide a type called Poly, which depends on another type, R say, specifying the ring you are working over. e.g. R might be the integers, or another polynomial ring, or a ring of matrices, etc.

Mathematics, especially algebra, is inherently dependent, meaning that Julia's dependent type system is ideal for computer algebra.

2.0.5 Polymorphism

Julia is polymorphic, meaning that multiple functions (and operators) can have the same name. For example, one can write separate versions of the determinant function for matrices over a ring versus matrices over a field.

In most dynamic languages, e.g. Python, one writes a single function det(M), say, which may then need many lines of code to sort out what kind of object we are taking the determinant of. All of this decision code is executed at runtime and places an additional burden on the programmer.

The Julia dispatch mechanism, on the other hand, does all of this automatically, alleviating the need to write such decision code.

One can still write generic, catch-all functions, which work over any ring, say. But these do not first need to delegate to a whole bunch of special cases where specialised algorithms can be used.

This greatly improves clarity and readability of code, as specialised versions of generic functions are clearly delineated by their type signatures.

In addition to this, Julia allows functions to not only be overloaded by other Julia functions of the same name when certain types of rings are passed as parameters, but one can also overload functions and operators with highly specialised C/C++ implementations via the Julia native code interface.

2.0.6 Abstract types and genericity

Julia doesn't provide classes a la C++. As such, it is not possible to inherit data types. Instead Julia achieves genericity by inheriting behaviour.

This is achieved by providing abstract types (also known as type classes).

Each type belongs to an *abstract type*, which can be thought of as a collection of types that have some common behaviour. For example, all the different native integer types in Julia belong to a common `Integer` type class. Arithmetic operations and greatest common divisor functions make sense for all of these integer types, for example.

Making all the native integer types belong to a single abstract `Integer` type class means that it is possible to write a function that accepts any native integer type as input simply by specifying that the given parameter is an `Integer`. That function will then accept any values whose type belongs to `Integer`.

Abstract types can also belong to one another. For example, the `Integer` abstract type belongs to the `Real` abstract type in Julia.

To Julia this means that any type which belongs to `Integer` is more specific than a type which belongs to `Real` but not to `Integer`.

This gives Julia a very powerful means of providing genericity. One can provide two versions of a function, one which works only with an `Integer` and one which works with any `Real`.

If a machine integer is passed to such a generic function, Julia will determine the most specific version of the function that applies. In this case the `Integer` version would be called.

If some other number is passed, which is not an integer, Julia falls back to the less specific version, namely the `Real` version of the function.

Of course, one can also provide an even more specific implementation which only works for one specific type, as opposed to an entire type class. Julia will take this as being even more specific and will call this preferentially over the less specific versions of the function, if it applies.

2.1 Gotchas with Julia

Like all languages, Julia has some design decisions and gotchas that new users should be aware of. We provide an incomplete list of these below.

2.1.1 1-indexed arrays

All Julia arrays are indexed starting with 1, i.e. `A[1]` is the first entry, not `A[0]`.

This is a deliberate, long standing design decision in Julia. Many mathematical papers make use of 1-based indexing for matrices and the like.

Some other computer algebra packages, such as Pari/GP make use of the same convention as Julia.

Experience shows that it rarely takes more than a couple of days to get used to when coming from a language that uses the other convention.

2.1.2 Bignums

Julia provides a built-in multiple precision integer type (using GMP). However for performance it also provided native integer types.

In order to maximise performance, Julia doesn't automatically switch from native integers to bignums on overflow. This means that if one starts off with native integers, the computation will only switch to bignums if the programmer does that explicitly.

The main place this causes an issue is when using constant integers at the REPL. If the integers are small enough to fit in a native machine word, then that is what will be used. If they are too large an `Int128` will be used if that is big enough, otherwise a multiple precision `BigInt` will be used.

This means that one must tell Julia explicitly what kind of integer one wants, if the default is not desired. This is similar to just about any statically typed language which provides bigints. The only ways around it would be for Julia to only provide bigints and no native types, or to require type annotation of all variables, either of which is not desirable.

2.1.3 Supertypes of core types

As we explained above, Julia provides native integer types, e.g. `Int` which belong to a single type class `Integer`.

In a computer algebra system, we'd really like `Integer` to then belong to some more general abstract type class, `RingElem` say.

Unfortunately, it is not possible to change the abstract type that Julia's own types and abstract types belong to.

This will be a gotcha for users of Nemo: Julia integers do not belong to Nemo's `RingElem` abstract type.

One way around this is to allow types to belong to a union type, `Union(Integer, RingElem)`. The other way around it is simply to implement two versions of a function, one which accepts an `Integer`, the other which accepts a *RingElem*.

Fortunately, one often wants to treat integers much more specially than general ring elements, so this doesn't cause too much of a problem in practice.

2.1.4 Division operators

The division functions in Julia are geared towards numerical, rather than algebraic, applications.

The main division operator `/` in Julia returns a floating point number when dividing integers and the `div` function treats even the rationals as a model of the reals, so that `div(a, b)` always returns the exact value a/b (in the mathematical sense), truncated to the nearest whole number in the direction of zero.

Julia also provides `//` for creating fraction objects, e.g. for creating rational fractions from integers.

For algebraic operations, Nemo introduces an additional `divexact` function, which assumes you are performing an exact division of two elements of a ring. We recommend using it exclusively, unless one knows that one wants precisely the Julia behaviour.

2.1.5 Multiple inheritance

The abstract type hierarchy in Julia is one of its best features. It really makes the language flexible and powerful.

However, at present it is only possible for a given type to belong to one abstract type (and transitively to any abstract types that it in turn belongs to).

There are cases where it would be useful if types could belong to multiple abstract types, e.g. `Inexact` and `RingElem` for elements that belong to inexact rings as opposed to exact rings.

There's currently no way around this in Julia, and so the abstract type hierarchy used by Nemo is pretty basic.

Instead of having a multitude of abstract types, we rely on the existence of functions for specific rings to determine what can be done with them.

For example, instead of having a `EuclideanDomainElem` abstract class for elements of a Euclidean domain, we rely on the existence of the `gcd` function for specific rings that happen to be Euclidean domains. If the `gcd` function doesn't exist, Julia will throw an exception if you try to call a function on that ring that requires a greatest common divisor to be defined. This (mostly) works well in practice.

2.1.6 Performance gotchas

There are a number of cases where Julia will provide very poor performance, which often catch new implementors unawares.

Firstly, Julia doesn't do jit compilation at the top level. If you simply type a series of isolated statements into the REPL they will be interpreted, not jit compiled.

If you want to get good performance, you should put your code inside a function and call it. The jit compilation happens at the function boundary and so if your code is inside a function it will execute quickly.

Top level variables are also very slow to access in Julia. This is because they could change type at any time and Julia can have no way to predict this. Therefore they aren't as fast as local variables in functions. Type inference can guarantee that local variables always have the same type meaning that faster code can result.

Julia provides the keyword `const` for top level constants. The value of such constants can't subsequently be changed, but Julia can then rely on them having the same type (and value), and they do not suffer from the same performance issues as top level variables.

3 Installing Nemo

There are currently two steps to installing Nemo: install Julia-0.4 and use the Julia package system to install Nemo.

To install Julia, please refer to the installation instructions on the Julia website (http://julialang.org/).

On Ubuntu, if you have root access, you can simply add a ppa and use apt-get. E.g. on my system it was as follows

```
sudo add-apt-repository ppa:staticfloat/juliareleases
sudo apt-get update
sudo apt-get install julia
```

Once Julia-0.4 is installed, start up Julia (type `julia` at the command line/shell prompt).

Now you can clone, build and test Nemo as follows:

```
Pkg.clone("https://github.com/wbhart/Nemo.git")
Pkg.build()
Pkg.test("Nemo")
```

The tests should only take a few seconds to a minute to run per module, depending on the speed of your machine.

If you encounter bugs in Nemo, please report them to nemo-devel@googlegroups.com.

4 Using Nemo

Julia has a module system, and we use it to provide access to Nemo.

At present there is a single module, called `Nemo`. To import it and use all exported functionality simply type

```
using Nemo
```

5 Nemo fundamentals

Nemo provides both generic algorithms and types, and wrappers of implementations and native types provided by various C/C++ libraries, such as Flint, Pari, Antic, etc.

For example, there are Flint polynomial types, Pari polynomials types, etc., but also a generic polynomial type whose coefficients can be in any Nemo ring (e.g. another polynomial ring, or a residue ring, etc.).

Many of the generic algorithms in Nemo are specialised for working over fields. For example, different algorithms are used when computing greatest common divisors of polynomials over more general rings (where it makes sense) and polynomials over fields.

Some Nemo objects don't belong to either rings or fields, e.g. general $m \times n$ matrices belong to matrix spaces, which are neither rings nor fields.

5.1 Parent objects and Nemo type classes

We call the ring or field or space that a Nemo arithmetic object belongs to, its *parent*.

Parents are implemented in Nemo as objects. For example, there is an object corresponding to the ring of integers as implemented by Flint, namely the unique parent object of type `FlintIntegerRing`.

A function named `parent` can be used to determine the parent of any arithmetic object in Nemo. In many cases, the arithmetic objects themselves contain pointers to their parent object.

A parent can be thought of as a kind of *mathematical type*. For example, elements of $\mathbb{Z}/5/Z$ and elements of $\mathbb{Z}/7\mathbb{Z}$ have the same Julia type but different parents. The Julia type only encodes the fact that we are working in a residue ring of the integers, but the parents encode the fact that we are working in two different rings $\mathbb{Z}/5\mathbb{Z}$ and $\mathbb{Z}/7\mathbb{Z}$. One of the parent objects will contain the modulus 5, the other will contain the modulus 7.

Both Nemo arithmetic objects and parent objects have types, and these types belong to various abstract types defined by Nemo.

At the highest level of the Nemo type hierarchy are the abstract types for the parent objects, namely `Collection`, `Ring` and `Field`. Corresponding to these are the abstract types for the corresponding arithmetic objects themselves: `CollectionElem`, `RingElem`, `FieldElem`.

Thus, a polynomial would have a type that belonged to `RingElem` and its parent would have a type that belonged to `Ring`.

Actually, the abstract type hierarchy is a bit richer than this. In fact we have a number of more refined abstract types: `PolyElem`, `SeriesElem` and `ResidueElem` are abstract types that all belong to `RingElem`; `FractionElem` is an abstract type that belongs to `FieldElem`; and `MatElem` belongs to `CollectionElem`.

So in fact, polynomials have types that belong to `PolyElem` which in turn belongs to `RingElem`.

5.2 Basic integer and rational rings

Before one can build generic rings over other rings, one needs a starting point on which to build.

For this purpose, Nemo provides wrappers of various C libraries, such as Flint and Pari, which have implementations of basic rings such as the integers and rationals.

Nemo provides some convenient names for the parent objects for these basic rings. Thus, `ZZ` is the name of the parent object for the ring of integers and `QQ` is the parent object for the rationals.

Parent objects are callable in Nemo. They can be used to create objects with that parent. For example, `ZZ(2)` creates the integer 2 and `QQ(2, 3)` creates the rational 2/3.

By default, Nemo makes ZZ an alias for the unique parent object of type FlintIntegerRing and QQ is an alias for the unique parent object of type FlintRationalField.

Once we have these basic rings we can build up more generic rings with these as base.

Flint also provides other basic rings such as padic rings and finite fields.

5.3 Nemo domains

As ZZ and QQ are just variables, they can be reassigned by the user. Therefore one can change ZZ to be the unique object of type PariIntegerRing or QQ to be the unique object of type PariRationalField. If one then builds up a generic ring over ZZ for example, it would then be implemented using Pari integers instead of Flint integers.

In Nemo we call the underlying system that is used to implement the basic rings the domain. One might select a different domain to the default one for performance reasons or because of the available functionality for that domain, or maybe because one wishes to work predominantly with one package instead of another.

This system of domains gives the user good control over the implementation used and allows the user to more easily tell which external library is responsible for the underlying implementation of their code. This allows the user to more easily credit (or blame) external implementations.

Currently, for ease of use, we do allow maximal orders of number fields to be built from Antic number fields, even though Pari provides the underlying implementation of maximal orders and their ideals. This reflects the fact that Antic is very new and currently doesn't perform such computations.

Below we discuss each of the parent types and object types available in Nemo and list all the functions available for each, with examples.

6 Generic rings

Internally, Nemo defines an abstract type for rings, with the following line of code:

```
abstract Ring
```

You can check (or assert) that a given type belongs to Ring with the <: operator. For example, we can check that the type of Nemo's ZZ parent object belongs to Ring

```
typeof(ZZ) <: Ring
```

Nemo will return true, indicating that the type of ZZ does belong to Ring.

Note that <: can only be used to determine if a type (or abstract type) belongs to an abstract type. Thus ZZ does not belong to Ring since ZZ is not a type but a mathematical parent object.

An element of a ring has a type that belongs to RingElem. For example, polynomials and integers have types that belong to RingElem.

For example the following code will return true:

```
typeof(ZZ(1)) <: RingElem
```

6.1 Generic polynomial rings ($R[x]$) : Poly{T}

Nemo has a parametric polynomial type (called Poly) for generic polynomials over a ring T. There's not any need to deal directly with the Poly type, as we provide a function PolynomialRing for constructing it.

The Poly type takes a parameter, namely the type of ring element that the coefficients will have, e.g. if there was no special type for Flint polynomials over the integers, the type of polynomials over the integers would be Poly{fmpz}.

Generic polynomials have parent PolynomialRing, which is also parameterised by the type of the ring element of the polynomial coefficients, the same as the Poly type is.

We have that PolynomialRing belongs to the Ring type class and Poly belongs to the PolyElem type class which in turn belongs to the RingElem type class.

To get the parent object for the coefficient ring we can call the base_ring function. It accepts either a polynomial or a polynomial parent object.

We can also get a symbol representing the name of the variable (as it will print) of a polynomial ring.

For example we have:

```
R, x = PolynomialRing(ZZ, "x")
S, y = PolynomialRing(R, "y")

base_ring(S) == R
var(S) == :y
```

6.1.1 Constructors

Because dealing with the Julia types directly is somewhat messy, we provide a simple function to create the parent of a polynomial in Nemo.

```
PolynomialRing{T <: Ring}(::T, s::String)
```

> This function takes a parent object which belongs to the Ring class (e.g. T = ZZ), specifying the ring that the polynomial coefficients are to belong to, and a String giving the string representation of a variable (e.g. "x"). It returns a tuple (R, x) consisting of the parent of the polynomial ring being constructed, R, and the degree 1 polynomial x in this ring.

This is easier to understand by giving an example

```
R, x = PolynomialRing(ZZ, "x")
S, y = PolynomialRing(R, "y")

f = (x^2 + 2x + 1)*y^3 + (x - 1)*y + 4x - 1
```

Notice that we can build up polynomials from the degree one polynomials x and y returned by the calls to PolynomialRing.

Another thing to note is that there is no reason why the variable name x has to be the same as the string "x". The latter is simply how x will print when we evaluate it. Of course, in most situations it is convenient to use the same letter for the variable and for the string.

Once we have constructed a polynomial ring as above, we can use it to construct polynomials of various kinds.

This is achieved by making the parent object of the polynomial ring callable. In the following we assume that S is a parent object of a polynomial ring, e.g. created by the following:

```
R, x = PolynomialRing(ZZ, "x")
S, y = PolynomialRing(R, "y")
```

The parent object `S` is made callable in Nemo by overloading the Julia `call` operator for polynomial ring parent objects. The type `T` below will refer to the type of the polynomial coefficients.

`S()`

Construct a polynomial of length 0 in the given polynomial ring.

`S(a::Integer)`
`S(a::T)`

Construct a polynomial of length 1 whose constant coefficient is a if $a \neq 0$, otherwise construct the zero polynomial.

`S(a::Array{T, 1})`

Construct a polynomial whose coefficients are given by the elements of the array `a`, starting with the constant coefficient at index 1. The array must be fully initialised, otherwise an exception may result.

`S(a::Poly{T})`

Return a reference to the polynomial `a`.

`S{R <: Ring}(a::R)`

Try to convert the value `a` to the polynomial type parameterised by `T` and the symbol, if possible, and return the value, else raise an error. This constructor is used to coerce polynomials and coefficients from subordinate rings up into the polynomial ring.

Examples.

Here are some examples of these constructors in action. Note that we don't actually ever need to deal with the parameterised `Poly` type directly.

```
R, x = PolynomialRing(ZZ, "x")
S, y = PolynomialRing(R, "y")
T, z = PolynomialRing(S, "z")

f = x^2 + y^3 + z + 1

g = S(2)

h = S(x^2 + 2x + 1)

j = T(x + 2)

k = S([x, x + 2, x^2 + 3x + 1])

l = S(k)
```

6.1.2 Basic manipulation

`degree{T <: RingElem}(a::Poly{T})`

Return the degree of the polynomial a. We define the degree of the zero polynomial to be -1.

`length{T <: RingElem}(a::Poly{T})`

Return the length of the polynomial a, i.e. its degree plus one.

`normalise{T <: RingElem}(a::Poly{T}, len::Int)`

Return the normalised length of the polynomial a assuming its unnormalised length is `len`. A polynomial is normalised if it is either length 0 or its leading coefficient is nonzero. This function is usually used internally.

`coeff{T <: RingElem}(a::Poly{T}, n::Int)`

Return the coefficient of the degree n term of the polnomial a, or zero if it has none.

`lead{T <: RingElem}(a::Poly{T})`

Return the leading coefficient of the polynomial a, or zero if a has length 0.

`iszero{T <: Ring}(a::Poly{T})`

Return `true` if the polynomial a is the additive identity of the polynomial ring, otherwise return `false`.

`isone{T <: Ring}(a::Poly{T})`

Return `true` if the polynomial a is the multiplicative identity of the polynomial ring, otherwise return `false`.

`isgen{T <: Ring}(a::Poly{T})`

Return `true` if the polynomial a is the generator (variable) of the polynomial ring, otherwise return `false`.

`isunit{T <: Ring}(a::Poly{T})`

Return `true` if the polynomial a is invertible, i.e. is of length 1 and has invertible constant coefficient, otherwise return `false`.

`zero{T <: Ring}(::Type{Poly{T}})`

Return the additive identity 0 in the given polynomial ring.

`one{T <: Ring}(::Type{Poly{T}})`

Return the multiplicative identity 1 in the given polynomial ring.

```
gen{T <: Ring}(::Type{Poly{T}})
```

Return the generator (variable) of the given polynomial ring.

```
canonical_unit{T <: Ring}(a::Poly{T})
```

Used for canonicalising fractions. The function simply returns `canonical_unit` of the leading coefficient.

```
deepcopy{T <: Ring}(a::Poly{T})
```

Make a new polynomial which is arithmetically equal to a.

Examples.

Here are some examples of the basic manipulation functions.

```
R, x = PolynomialRing(ZZ, "x")
S, y = PolynomialRing(R, "y")

a = zero(S)
b = one(S)
c = gen(S)
d = isunit(b)

f = x^2*y + 2x + 1
g = lead(f)

h = x*y^2 + (x + 1)*y + 3
j = coeff(h, 2)

if isgen(y)
   println(y, " is the generator of the ", parent(y))
end

k = isone(b)
m = iszero(a)

n = canonical_unit(-x*y + x + 1)
```

6.1.3 Binary operators

The following binary operators are provided for polynomials. Note that if both operands are not in the same polynomial ring, Nemo will try to coerce them into one of the two rings before applying the operation.

```
+{T <: RingElem}(a::Poly{T}, b::Poly{T})
-{T <: RingElem}(a::Poly{T}, b::Poly{T})
*{T <: RingElem}(a::Poly{T}, b::Poly{T})
```

Examples.

Here are some examples of the binary operators.

```
R, x = PolynomialRing(ZZ, "x")
S, y = PolynomialRing(R, "y")

f = x*y^2 + (x + 1)*y + 3
g = (x + 1)*y + (x^3 + 2x + 2)

h = f - g
n = f*g
p = f + g
```

6.1.4 Ad hoc binary operators

The following ad hoc binary operators are faster than first coercing all operands into the polynomial ring.

```
*{T <: RingElem}(x::Poly{T}, y::Int)
*{T <: RingElem}(x::Poly{T}, y::fmpz)
*{T <: RingElem}(x::Int, y::Poly{T})
*{T <: RingElem}(x::fmpz, y::Poly{T})
```

Examples.

Here are some examples of the ad hoc binary operators.

```
R, x = PolynomialRing(ZZ, "x")
S, y = PolynomialRing(R, "y")

f = x*y^2 + (x + 1)*y + 3
g = (x + 1)*y + (x^3 + 2x + 2)
h = g - 4
k = fmpz(5) - g
l = f*7
```

6.1.5 Comparison

```
==(T <: RingElem}(x::Poly{T}, y::Poly{T})
```

Julia automatically provides a `!=` operator.

`isequal(x::Poly{T}, y::Poly{T})`

Returns `true` if `x == y` and if each coefficient of `x` matches the corresponding coefficient of `y` according to `isequal`, when applied recursively, otherwise return `false`.

Examples.

Here are some examples of comparison.

```
R, x = PolynomialRing(ZZ, "x")
S, y = PolynomialRing(R, "y")

f = x*y^2 + (x + 1)*y + 3
g = x*y^2 + (x + 1)*y + 3

if f == g
   println("f is equal to g")
end
```

6.1.6 Ad hoc comparison

The following comparison operators are faster than first coercing the arguments into one or the other ring.

```
==(T <: RingElem}(x::Poly{T}, y::Integer)
==(T <: RingElem}(x::Integer, y::Poly{T})
```

Examples.

Here is an example of ad hoc comparison.

```
R, x = PolynomialRing(ZZ, "x")
S, y = PolynomialRing(R, "y")

if S(1) == 1
   println("S(1) is equal to ZZ(1)")
end
```

6.1.7 Unary operators

```
-(T <: RingElem}(a::Poly{T})
```

Examples.

Here are some examples of unary operators.

```
R, x = PolynomialRing(ZZ, "x")
S, y = PolynomialRing(R, "y")

f = x*y^2 + (x + 1)*y + 3
g = -f
```

6.1.8 Truncation

`truncate(T <: RingElem}(a::Poly{T}, n::Int)`

Return the polynomial a truncated to length n. If $n < 0$ we throw a `DomainError()`.

`mullow(T <: RingElem}(a::Poly{T}, b::Poly{T}, n::Int)`

Return the polynomial `a*b` truncated to length n. If $n < 0$ we throw a `DomainError()`.

Examples.

Here are some examples of truncated operations.

```
R, x = PolynomialRing(ZZ, "x")
S, y = PolynomialRing(R, "y")

f = x*y^2 + (x + 1)*y + 3
g = (x + 1)*y + (x^3 + 2x + 2)

h = truncate(f, 1)
k = mullow(f, g, 4)
```

6.1.9 Reversal

`reverse{T <: RingElem}(x::Poly{T}, len::Int)`

Return the reverse of the polynomial x, thought of as a polynomial of the given length (the polynomial will be notionally truncated or padded with zeroes before the leading term if necessary to match the specified length). The resulting polynomial is normalised. If `len` is negative we throw a `DomainError()`.

`reverse{T <: RingElem}(x::Poly{T})`

Return the reverse of the polynomial x, i.e. the leading coefficient of x becomes the constant coefficient of the result, etc. The resulting polynomial is normalised.

Examples.

Here are some examples of reversal.

```
R, x = PolynomialRing(ZZ, "x")
S, y = PolynomialRing(R, "y")

f = x*y^2 + (x + 1)*y + 3

g = reverse(f, 7)
```

6.1.10 Shifting

`shift_left{T <: RingElem}(x::Poly{T}, n::Int)`

Return the polynomial that results from shifting the coefficients of the polynomial x to the left by n places, i.e. $f \times x^n$ where x is the generator of the polynomial ring that f belongs to. If $n < 0$ we throw a `DomainError()`.

`shift_right{T <: RingElem}(x::Poly{T}, n::Int)`

Return the polynomial that results from shifting the coefficients of the polynomial x to the right by n places, i.e. by dividing f by x^n and throwing away the remainder. If n is greater than or equal to the length of x then the zero polynomial results. If $n < 0$ we throw a `DomainError()`.

Examples.

Here are some examples of shifting.

```
R, x = PolynomialRing(ZZ, "x")
S, y = PolynomialRing(R, "y")

f = x*y^2 + (x + 1)*y + 3

g = shift_left(f, 7)
h = shift_right(f, 2)
```

6.1.11 Powering

```
^{T <: RingElem}(a::Poly{T}, n::Int)
```

Return the polynomial a^n. If $n < 0$ we throw a `DomainError()`.

Examples.

Here are some examples of powering.

```
R, x = PolynomialRing(ZZ, "x")
S, y = PolynomialRing(R, "y")

f = x*y^2 + (x + 1)*y + 3
g = f^5
```

6.1.12 Modular arithmetic

When we have polynomials over residue rings or fields, we can reduce modulo a polynomial over the base ring. In this way we can do modular arithmetic with polynomials.

```
mulmod{T <: Union(ResidueElem, FieldElem)}(a::Poly{T},
                                           b::Poly{T}, d::Poly{T})
```

Return $ab \pmod{d}$.

```
invmod{T <: Union(ResidueElem, FieldElem)}(a::Poly{T}, d::Poly{T})
```

Return $a^{-1} \pmod{d}$. If an impossible inverse is encountered, an exception is thrown.

```
powmod{T <: Union(ResidueElem, FieldElem)}(a::Poly{T}, b::Int,
                                           d::Poly{T})
```

Return $a^b \pmod{d}$.

Here are some examples of modular arithmetic.

```
R, x = PolynomialRing(QQ, "x")
S = ResidueRing(R, x^3 + 3x + 1)
T, y = PolynomialRing(S, "y")

f = (3*x^2 + x + 2)*y + x^2 + 1
g = (5*x^2 + 2*x + 1)*y^2 + 2x*y + x + 1
h = (3*x^3 + 2*x^2 + x + 7)*y^5 + 2x*y + 1

invmod(f, g)
mulmod(f, g, h)
powmod(f, 3, h)
```

6.1.13 Exact division

`divexact{T <: RingElem}(a::Poly{T}, b::Poly{T})`

Return a divided by b. The result is only meaningful if the division is exact. If $b = 0$ we throw a `DivideError()`.

If `T` is a residue ring and the leading coefficient of b is not invertible in the `T`, an error will be thrown.

Examples.

Here is an example of exact division.

```
R, x = PolynomialRing(ZZ, "x")
S, y = PolynomialRing(R, "y")

f = x*y^2 + (x + 1)*y + 3
g = (x + 1)*y + (x^3 + 2x + 2)

h = divexact(f*g, f)
```

6.1.14 Ad hoc exact division

`divexact{T <: RingElem}(a::Poly{T}, b::T)`
`divexact{T <: RingElem}(a::Poly{T}, b::Integer)`

Return a divided by b. The result is only meaningful if the division is exact. If $b = 0$ we throw a `DivideError()`.

Examples.

Here are some examples of ad hoc exact division.

```
R, x = PolynomialRing(ZZ, "x")
S, y = PolynomialRing(R, "y")

f = x*y^2 + (x + 1)*y + 3
g = (x + 1)*y + (x^3 + 2x + 2)

h = divexact(3*f, 3)
k = divexact(x*g, x)
```

6.1.15 Euclidean division

Over a residue ring we can define Euclidean division.

```
mod{T <: Union(ResidueElem, FieldElem)}(f::Poly{T}, g::Poly{T})
```

> Return the Euclidean remainder of f divided by g, i.e. r such that $a = bq + r$ for some polynomial r with $\deg(r) < \deg(b)$. If $g = 0$ we throw a `DivideError()`.

```
divrem{T <: Union(ResidueElem, FieldElem)}(f::Poly{T}, g::Poly{T})
```

> Return the Euclidean quotient and remainder of f divided by g, i.e. a tuple `(q, r)` such that $a = bq + r$ with $\deg(r) < \deg(b)$. If $g = 0$ we throw a `DivideError()`.

Examples.

Here are some examples of Euclidean division.

```
R = ResidueRing(ZZ, 7)
S, x = PolynomialRing(R, "x")
T = ResidueRing(S, x^3 + 3x + 1)
U, y = PolynomialRing(T, "y")

f = y^3 + x*y^2 + (x + 1)*y + 3
g = (x + 1)*y^2 + (x^3 + 2x + 2)

h = mod(f, g)
q, r = divrem(f, g)
```

6.1.16 Pseudodivision

Given two polynomials a, b, pseudodivision computes polynomials q and r with $\text{length}(r) < \text{length}(b)$ such that

$$L^d a = bq + r,$$

where $d = \text{length}(a) - \text{length}(b) + 1$ and L is the leading coefficient of b.

We call q the pseudoquotient and r the pseudoremainder.

```
pseudorem{T <: RingElem}(a::Poly{T}, b::Poly{T})
```

> Return the pseudoremainder of a divided by b. If $b = 0$ we throw a `DivideError()`.

```
pseudodivrem{T <: RingElem}(a::Poly{T}, b::Poly{T})
```

> Return a tuple (q, r) consisting of the pseudoquotient and pseudoremainder of a divided by b. If $b = 0$ we throw a `DivideError()`.

Examples.

Here are some examples of pseudodivision.

```
R, x = PolynomialRing(ZZ, "x")
S, y = PolynomialRing(R, "y")

f = x*y^2 + (x + 1)*y + 3
g = (x + 1)*y + (x^3 + 2x + 2)

h = pseudorem(f, g)
q, r = pseudodivrem(f, g)
```

Note that pseudodivision is not always useful. In general it computes a multiple of the quotient, and for example in the case of polynomials over a residue ring, this may lead to meaningless results in the presence of zero divisors. For example

```
R = ResidueRing(ZZ, 6)
S, x = PolynomialRing(R, "x")

f = 3*x^5 + x^4 + 3*x^2 + 2
g = 2*x^3 + 5*x^2 + 2*x + 2

pseudorem(f, g)
```

Here Nemo returns `3*x^2`, which happens to be `3*g` in this ring.

6.1.17 Content, primitive part, GCD and LCM

When the base ring provides a `gcd` function we can provide numerous GCD related functions for polynomials over that ring.

```
content{T <: RingElem}(a::Poly{T})
```

> Return the content of the polynomial a, i.e. the greatest common divisor of all its coefficients, if this exists.

```
primpart{T <: RingElem}(a::Poly{T})
```

> Return the primitive part of the polynomial a, i.e. the polynomial divided by its content, if it exists.

```
gcd{T <: RingElem}(a::Poly{T}, b::Poly{T})
```

> Return the greatest common divisor of the polynomials a and b if it exists.

```
lcm{T <: RingElem}(a::Poly{T}, b::Poly{T})
```

> Return the least common multiple of the polynomials a and b. We define $\text{lcm}(a, b) = ab/\text{gcd}(a, b)$.

We allow GCD for polynomials over residue rings, e.g. over $\mathbb{Z}/n\mathbb{Z}$, even when the residue ring has zero divisors. However, if an impossible inverse is encountered during the computation, due to zero divisors, we throw an error.

```
gcdinv{T <: Union(ResidueElem, FieldElem)}(a::Poly{T}, b::Poly{T})
```

Return a tuple (g, s) such that g is the gcd of a and b and s and t are polynomials such that $g = as + bt$. The value g will be 1 if a is invertible modulo b.

Examples.

Here are some examples of content, primitive part and GCD.

```
R, x = PolynomialRing(ZZ, "x")
S, y = PolynomialRing(R, "y")

k = x*y^2 + (x + 1)*y + 3
l = (x + 1)*y + (x^3 + 2x + 2)
m = y^2 + x + 1

n = content(k)
p = primpart(k*(x^2 + 1))
q = gcd(k*m, l*m)
r = lcm(k*m, l*m)

R, x = PolynomialRing(QQ, "x")
T = ResidueRing(R, x^3 + 3x + 1)
U, z = PolynomialRing(T, "z")

r = z^3 + 2z + 1
s = z^5 + 1
u, v = gcdinv(r, s)
```

6.1.18 Evaluation

```
evaluate{T <: RingElem}(a::Poly{T}, b::T)
evaluate{T <: RingElem}(a::Poly{T}, b::Integer)
evaluate{T <: RingElem}(a::Poly{T}, b::fmpz)
```

Return the value of the polynomial a evaluated at b.

Examples.

Here are some examples of evaluation.

```
R, x = PolynomialRing(ZZ, "x")
S, y = PolynomialRing(R, "y")

f = x*y^2 + (x + 1)*y + 3
g = evaluate(f, 3)
h = evaluate(f, x^2 + 2x + 1)
```

6.1.19 Composition

```
compose{T <: RingElem}(a::Poly{T}, b::Poly{T})
```

Return the composition of the polynomial a with b, i.e. $a \circ b$.

Examples.

Here are some examples of composition.

```
R, x = PolynomialRing(ZZ, "x")
S, y = PolynomialRing(R, "y")

f = x*y^2 + (x + 1)*y + 3
g = (x + 1)*y + (x^3 + 2x + 2)
h = compose(f, g)
```

6.1.20 Derivative

`derivative{T <: RingElem}(a::Poly{T})`
Return the derivative of the polynomial a with respect to its main variable.

Examples.

Here are some examples of computing derivatives.

```
R, x = PolynomialRing(ZZ, "x")
S, y = PolynomialRing(R, "y")

h = x*y^2 + (x + 1)*y + 3
j = derivative(h)
```

6.1.21 Integral

`integral{T <: Union(ResidueElem, FieldElem)}(f::Poly{T})`
Return the integral of the polynomial f. The constant coefficient of the result is taken to be 0.

Examples.

Here are some examples of computing integrals.

```
R, x = PolynomialRing(QQ, "x")
S = ResidueRing(R, x^3 + 3x + 1)
T, y = PolynomialRing(S, "y")

f = (x^2 + 2x + 1)*y^2 + (x + 1)*y - 2x + 4

g = integral(f)
```

6.1.22 Resultant

`resultant{T <: RingElem}(a::Poly{T}, b::Poly{T})`
Return the resultant of the polynomials a and b.

Examples.

Here are some examples of computing resultants.

```
R, x = PolynomialRing(ZZ, "x")
S, y = PolynomialRing(R, "y")

f = 3x*y^2 + (x + 1)*y + 3
g = 6(x + 1)*y + (x^3 + 2x + 2)
h = resultant(f, g)
```

6.1.23 Discriminant

`discriminant{T <: RingElem}(a::Poly{T})`

Compute the discriminant of the polynomial a.

Examples.

Here are some examples of computing discriminants.

```
R, x = PolynomialRing(ZZ, "x")
S, y = PolynomialRing(R, "y")

f = x*y^2 + (x + 1)*y + 3
g = discriminant(f)
```

6.1.24 Bezout identity

The Bezout identity is the equivalent, for polynomials, of the extended GCD in the case of the integers. Given polynomials a, b, we can find polynomials s, t such that

$$sa + bt = r,$$

where r is the resultant of a and b.

`bezout{T <: RingElem}(a::Poly{T}, b::Poly{T})`

Return a tuple (r, s, t) such that r is the resultant of a and b and with polynomials s, t such that $r = as + bt$.

Examples.

Here are some examples of computing the Bezout identity.

```
R, x = PolynomialRing(ZZ, "x")
S, y = PolynomialRing(R, "y")

f = 3x*y^2 + (x + 1)*y + 3
g = 6(x + 1)*y + (x^3 + 2x + 2)
r, s, t = gcdx(f, g)
```

6.1.25 Special polynomials

The following functions compute univariate special polynomials. It is expected that the final argument in each of these function is the generator of the polynomial ring.

```
chebyshev_t{T <: RingElem}(n::Int, x::Poly{T})
```

Return the Chebyshev polynomial of the first kind $T_n(x)$, defined by $T_n(x) = \cos(n \cos^{-1}(x))$.

```
chebyshev_u{T <: RingElem}(n::Int, x::Poly{T})
```

Return the Chebyshev polynomial of the first kind $U_n(x)$, defined by $(n+1)U_n(x) = T'_{n+1}(x)$.

Examples.

Here are some examples of special polynomials.

```
R, x = PolynomialRing(ZZ, "x")
S, y = PolynomialRing(R, "y")

f = chebyshev_t(20, y)
g = chebyshev_u(15, y)
```

6.2 Generic residue rings $(R/(f))$: `Residue{T}`

Nemo allows the construction of residue rings.

There are two examples to keep in mind:

1. $\mathbb{Z}/n\mathbb{Z}$ for some positive integer n
2. $\mathbb{Q}[x]/(f)$ where f is a polynomial in $\mathbb{Q}[x]$.

Currently we only allow construction of residue rings of the form $R/(r)$ where R is a ring and (r) is a principal ideal generated by an element $r \in R$.

In the first example above the principal ideal is (n) and in the second the principal ideal is (f).

We don't require the ideal (r) to be maximal or prime (corresponding in the commutative case to residue rings that are fields and integral domains respectively).

Instead, we allow r to be any nonzero element of R, but raise an exception every time a computation in $R/(r)$ requires computing an impossible inverse.

The parent object for a residue ring in Nemo has type `ResidueRing` belonging to the type class `Ring` and residues have type `Residue` belonging to the type class `ResidueElem` which in turn belongs to `RingElem`.

There is no need to deal directly with these types, as we provide a function `ResidueRing` for constructing residue ring parent objects and various constructors for creating residue objects.

Both the `ResidueRing` and `Residue` types are parameterised by the type of objects in the base ring over which the residue ring is constructed.

Each `Residue` object contains a pointer to its parent object, which in turn contains the modulus for the residue.

6.2.1 Constructors

To make it easy to deal with the `ResidueRing` type in Nemo, we provide a function for constructing the parent object of a residue ring

```
ResidueRing{T <: RingElem}(S::Ring, el::T)
ResidueRing{T <: RingElem}(S::Ring, el::Integer)
```

> This function takes a base ring `S` and an element `el` of that ring. The function returns a parent object for the residue ring `S/(el)`.

In the constructors below we assume `R` has been created using the `ResidueRing` constructor, e.g:

```
R = ResidueRing(ZZ, 17)
```

We now discuss the various constructors available to create `Residue` objects that belong to the given ring. In each case, `T` is the type of elements of the base ring of the residue ring.

```
R(a::T)
R(a::Integer)
```

> Create a residue congruent to a. The value a is first reduced modulo the modulus of the residue ring.

```
R(a::Residue{T})
```

> Return a reference to the residue a. No copy of the data is made.

```
Residue()
```

> Create a new residue, which is congruent to 0.

```
Residue(a::RingElem)
```

> Try to coerce a into the base ring of the residue ring and then create a new residue congruent to it.

Examples.

Here are some examples of constructing residue rings and elements in them.

```
R = ResidueRing(ZZ, 16453889)

f = R(123)
g = R(f)
h = R(fmpz(12))
k = R()

S, x = PolynomialRing(R, "x")
T = ResidueRing(S, x^3 + 3x + 1)

m = T(x^4)
```

6.2.2 Basic manipulation

`modulus{T <: RingElem}(R::ResidueRing{T})`
`modulus{T <: RingElem}(a::Residue{T})`

Return the modulus of the given residue ring or of the parent of the given residue element respectively.

`data{T <: RingElem}(a::Residue{T})`

Return the value stored in the given residue element as an element of the base ring.

`zero{T <: RingElem}(R::ResidueRing{T})`

Return the additive identity of the given residue ring.

`one{T <: RingElem}(::ResidueRing{T})`

Return the multiplicative identity of the given residue ring.

`iszero{T <: RingElem}(a::Residue{T})`

Return `true` if the residue a is the additive identity in the residue ring, otherwise return `false`.

`isone{T <: RingElem}(a::Residue{T})`

Return `true` if the residue a is the multiplicative identity in the residue ring, otherwise return `false`.

`isunit{T <: RingElem}(a::Residue{T})`

Return `true` if the residue a is invertible in the residue ring, otherwise return `false`.

`canonical_unit{T <: RingElem}(a::Residue{T})`

Used for canonicalising fractions. The function simply returns a. No check is performed to check that a is actually invertible.

`deepcopy{T <: RingElem}(a::Residue{T})`

Create a new residue element which is arithmetically equal to the given residue element.

Examples.

Here are some examples of basic manipulation of residue rings.

```
R = ResidueRing(ZZ, 16453889)

f = modulus(R)
g = zero(R)

S, x = PolynomialRing(R, "x")
T = ResidueRing(S, x^3 + 3x + 1)

h = one(T)

k = isunit(h)

m = canonical_unit(R(11))
n = canonical_unit(T(x + 1))

p = isone(h)
q = iszero(g)
```

6.2.3 Unary operations

`-{T <: RingElem}(a::Residue{T})`
Return $-a$.

Examples.
Here are some examples of unary operators.

```
R = ResidueRing(ZZ, 16453889)

f = -R(12345)

S, x = PolynomialRing(R, "x")
T = ResidueRing(S, x^3 + 3x + 1)

h = -T(x^5 + 1)
```

6.2.4 Binary operators

The following binary operators for residue rings are provided.

`+{T <: RingElem}(a::Residue{T}, b::Residue{T})`

Return $a + b$.

`-{T <: RingElem}(a::Residue{T}, b::Residue{T})`

Return $a - b$.

`*{T <: RingElem}(a::Residue{T}, b::Residue{T})`

Return ab.

Examples.

Here are some examples of binary operators.

```
R = ResidueRing(ZZ, 12)

f = R(4)
g = R(6)

h = f + g
j = f - g
k = f*g

Q = ResidueRing(ZZ, 7)
S, x = PolynomialRing(Q, "x")
T = ResidueRing(S, x^3 + 3x + 1)

n = T(x^5 + 1)
p = T(x^2 + 2x + 1)

q = n + p
r = n - p
s = n*p
```

6.2.5 Exact division

`divexact{T <: RingElem}(a::Residue{T}, b::Residue{T})`

Return a divided by b. If an impossible inverse is encounted an exception is thrown.

Examples.

Here are some examples of exact division.

```
Q = ResidueRing(ZZ, 7)

a = Q(3)
b = Q(4)

m = divexact(a*b, a)

S, x = PolynomialRing(Q, "x")
T = ResidueRing(S, x^3 + 3x + 1)

n = T(x^5 + 1)
p = T(x^2 + 2x + 1)

t = divexact(n*p, p)

gcd{T <: RingElem}(a::Residue{T}, b::Residue{T})
```

6.2.6 GCD

Return the greatest common divisor of a and b.

Recall that a and b are represented by values a' and b' in the base ring. We define the greatest common divisor of a and b to be the residue class of $\gcd(\gcd(a', b'), m)$ when that is defined, where m is the modulus of the residue ring.

This definition does not depend on the choice of representatives of a and b.

Examples.

Here are some examples of GCD.

```
R = ResidueRing(ZZ, 12)

f = R(4)
g = R(6)

l = gcd(f, g)

Q = ResidueRing(ZZ, 7)
S, x = PolynomialRing(Q, "x")
T = ResidueRing(S, x^3 + 3x + 1)

n = T(x^5 + 1)
p = T(x^2 + 2x + 1)

u = gcd(n, p)
```

6.2.7 Ad hoc binary operators

In a residue ring R over a ring T, with modulus m, it is convenient to think of $n \in \mathbb{Z}$ as corresponding to 1 added to itself n times in T then reduced modulo m.

This allows us to efficiently define the following ad hoc operators, which effectively make the integers $n \in \mathbb{Z}$ a convenient notation for certain elements of the residue ring R.

```
+{T <: RingElem}(a::Residue{T}, b::Integer)
+{T <: RingElem}(a::Integer, b::Residue{T})
```

Return $a + b$ with the integer operand thought of as an element of the given residue ring.

```
-{T <: RingElem}(a::Residue{T}, b::Integer)
-{T <: RingElem}(a::Integer, b::Residue{T})
```

Return $a - b$ with the integer operand thought of as an element of the given residue ring.

```
*{T <: RingElem}(a::Residue{T}, b::Integer)
```

Return ab, i.e. the element a added to itself b times.

```
*{T <: RingElem}(a::Integer, b::Residue{T})
```

Return ab, i.e. the element b added to itself a times.

Examples.

Here are some examples of ad hoc binary operators.

```
R = ResidueRing(ZZ, 7)

a = R(3)

b = a + 3
c = 3 - a
d = 5a

S, x = PolynomialRing(R, "x")
T = ResidueRing(S, x^3 + 3x + 1)

f = T(x^5 + 1)
g = f + 4
h = 4 - f
k = f*5
```

6.2.8 Comparison

`==(T, S)(x::Residue{T}, y::Residue{T})`

Return `true` if $x = y$ arithmetically in the given residue ring, otherwise return false.

Julia automatically defines a `!=` operator.

`isequal(x::Residue{T}, y::Residue{T})`

Returns `true` if the residues (as elements of the base ring) compare equal recursively according to `isequal`, otherwise return `false`.

Examples.

Here are some examples of comparisons.

```
R = ResidueRing(ZZ, 7)

a = R(3)
b = a
c = R(2)

b == a
c != a

S, x = PolynomialRing(R, "x")
T = ResidueRing(S, x^3 + 3x + 1)

f = T(x^5 + 1)
```

```
g = 8f
h = f + g

f == g
h != g
isequal(f, g)
```

6.2.9 Ad hoc comparison

Thinking of integers as corresponding to elements of a residue ring, we can define the following ad hoc comparison operators.

`==(T, S)(x::Residue{T}, y::Integer)`

Given $x \in R$ for some residue ring R, return `true` if $x = y.1$ in R, otherwise return `false`.

`==(T, S)(x::Integer, y::Residue{T})`

Given $y \in R$ for some residue ring R, return `true` if $x.1 = y$ in R, otherwise return `false`.

Julia automatically provides corresponding `!=` operators.

Examples.

Here are some examples of ad hoc comparisons.

```
R = ResidueRing(ZZ, 7)

a = R(3)

a == 3
4 != a

S, x = PolynomialRing(R, "x")
T = ResidueRing(S, x^3 + 3x + 1)

f = T(x^5 + 1)

f != 5
```

6.2.10 Powering

`^(T <: RingElem)(a::Residue{T}, b::Int)`

Return a^b.

Examples.

Here are some examples of powering.

```
R = ResidueRing(ZZ, 7)

a = R(3)

b = a^5

S, x = PolynomialRing(R, "x")
T = ResidueRing(S, x^3 + 3x + 1)

f = T(x^5 + 1)

g = f^100
```

6.2.11 Inversion

So long as we don't encounter an impossible inverse, we can invert an element in a residue ring $R = T/(m)$, if T is a Euclidean ring, i.e. if we have a `gcd` function in T.

```
inv{T <: RingElem}(a::Residue{T})
```

> Return the multiplicative inverse of a in the given residue ring, i.e. return an element b such that $ab = 1$ in the residue ring, if such exists. If an impossible inverse is encountered during the computation, we throw an exception.

Examples.

Here are some examples of computing inverses.

```
R = ResidueRing(ZZ, 49)

a = R(5)

b = inv(a)

S, x = PolynomialRing(R, "x")
T = ResidueRing(S, x^3 + 3x + 1)

f = T(x^5 + 1)

g = inv(f)
```

6.2.12 Exact division

So long as we don't encounter an impossible inverse, we can divide elements in a residue ring $R = T/(m)$, if T is a Euclidean ring, i.e. if we have a `gcd` function in T.

```
divexact{T <: RingElem}(a::Residue{T}, b::Residue{T})
```

> Return the quotient of a by b in the given residue ring, i.e. return an element c such that $ac = b$ in the residue ring, if such exists. If an impossible inverse is encountered during the computation, we throw an exception.

Examples.

Here are some examples of exact division.

```
R = ResidueRing(ZZ, 49)

a = R(5)
b = R(3)

c = divexact(a, b)

S, x = PolynomialRing(R, "x")
T = ResidueRing(S, x^3 + 3x + 1)

f = T(x^5 + 1)
g = T(x^4 + x + 2)

h = divexact(f, g)
```

6.3 Generic power series rings ($R[[x]]$): PowerSeries{T}

In Nemo we provide univariate power series rings. The type of a parent object for a power series ring is a `PowerSeriesRing` which belongs to the `Ring` type class. Power series elements are of type `PowerSeries` which belongs to the `SeriesElem` type class, which in turn belongs to `RingElem`.

There's no need to deal directly with these types, as we provide a function `PowerSeriesRing` for constructing the ring parent objects and various constructors to create power series element objects.

As for the `Poly` type, the `PowerSeries` and `PowerSeriesRing` types take as parameters the type `T` of the elements in the base ring over which the power series ring is constructed.

For example, if there wasn't a special Flint type for power series, the type of a power series ring $\mathbb{Z}[[x]]$ over the Flint integers would be denoted `PowerSeriesRing{fmpz}` and elements of the ring would have type `PowerSeries{fmpz}`.

Power series differ from polynomials in Nemo primarily in that they have a precision attached to them. Each power series object has its own precision, stored in a field `prec` and each power series parent object has a maximum precision, stored in `prec_max`.

The precision is always a non-negative integer.

By default, power series in Nemo use a *capped relative* precision model. This means that the precision of a power series in a given ring is capped at the precision given by `prec_max`. Moreover, the precision of each individual power series is relative, meaning that if the leading term of a nonzero power series element is $c_a x^a$ and the precision is b then the power series is of the form $c_a x^a + c_{a+1}x^{a+1} + \ldots + O(x^{a+b})$.

The zero power series is simply taken to be $0 + O(x^b)$.

(Another commonly seen model of power series is the *capped absolute* model. In that case a precision b would indicate that all power series are of the form $c_0x^0 + c_1x + \ldots + O(x^b)$, where c_0 need not be nonzero. Currently Nemo doesn't provide an implementation of the capped absolute model.)

The capped relative model has the advantage that power series are stable multiplicatively. In other words, for nonzero power series f and g we have that `divexact(f*g, g) == f`.

However, capped relative power series are not additively stable, i.e. we do not always have $(f+g)-g = f$.

In the capped relative model we say that two power series are equal if they agree up to the *absolute* precision of the two power series. Thus, for example, $x^5 + O(x^10) == 0 + O(x^5)$, since the minimum absolute precision is 5.

During computations, it is possible for power series to "lose" precision due to cancellation. For example if $f = x^3 + x^5 + O(x^8)$ and $g = x^3 + x^6 + O(x^8)$ then $f - g = x^5 - x^6 + O(x^8)$ which now has relative precision 3 instead of relative precision 5.

Amongst other things, this means that equality is not transitive. For example $x^6 + O(x^1) == 0 + O(x^5)$ and $x^7 + O(x^12) == 0 + O(x^5)$ but $x^6 + O(x^1) \neq x^7 + O(x^12)$.

Sometimes it is necessary to compare power series not just for arithmetic equality, as above, but to see if they have precisely the same precision and terms. For this purpose we introduce the `isequal` function.

For example, if $f = x^2 + O(x^7)$ and $g = x^2 + O(x^8)$ and $h = 0 + O(x^2)$ then $f == g$, $f == h$ and $g == h$, but `isequal(f, g)`, `isequal(f, h)` and `isequal(g, h)` would all return `false`. However, if $k = x^2 + O(x^7)$ then `isequal(f, k)` would return `true`.

There are further difficulties if we construct polynomial over power series. For example, consider the polynomial in y over the power series ring in x over the rationals. Normalisation of such polynomials is problematic. For instance, what is the leading coefficient of $(0 + O(x^1))y + (1 + O(x^1))$?

If one takes it to be $(0 + O(x^1))$ then some functions may not terminate due to the fact that algorithms may require the degree of polynomials to decrease with each iteration. Instead, the degree may remain constant and simply accumulate leading terms which are arithmetically zero but not identically zero.

On the other hand, when constructing power series over other power series, if we simply throw away terms which are arithmetically equal to zero, our computations may have different output depending on the order in which power series are added!

One should be aware of these difficulties when working with power series. Power series, as represented on a computer, simply don't satisfy the axioms of a ring. They must be used with care in order to approximate operations in a mathematical power series ring.

Simply increasing the precision will not necessarily give a "more correct" answer and some computations may not even terminate due to the presence of arithmetic zeroes!

Note that power series in Nemo are currently *stored* in absolute rather than relative format. Thus $x^1 0 + O(x^3 0)$ has a length 11 polynomial underlying it, rather than a length 1 polynomial. This may change in a later version of Nemo.

6.3.1 Constructors

We define the following function for constructing the parent object of a power series ring.

```
PowerSeriesRing(R::Ring, p::Int, s::String)
```

> Returns a tuple `(S, x + O(x^p))` consisting of the parent `S` corresponding to the power series ring over the ring `R` and the generator `x + O(x^p)` of that power series ring, where p is the maximum precision of elements of the power series ring.
>
> The generator `x + O(x^p)` will be printed as per the supplied string `s`, which specifies the variable for the power series. This need not be the same as the variable name used to store the generator `x`, but it is usually convenient to make it to be so.

We also provide various constructors for creating elements of a power series ring. Below we assume that S is a power series ring parent object, created for example by the following code:

```
R, t = PolynomialRing(QQ, "t")
S, x = PowerSeriesRing(R, 30, "x")
```

We will also let p stand for the maximum precision, e.g. $p = 30$ in the case of S as we just defined it, and x to be the variable of the power series. We let T stand for the type of the coefficients of the power series.

`S()`

Create the power series $0 + O(x^p)$.

`S(a::Integer)`
`S(a::T)`

Create the power series $a + O(x^p)$.

`S(a::PowerSeries{T})`

Return a reference to the power series `a`. No copy of the data is made by this function.

`S(a::Array{T, 1}, n::Int p::Int)`

Create a power series with precision `p` and with coefficients given by the array `a`. The constant coefficient of the power series will be the first element in the array. The length `n` must be a nonnegative integer and may not exceed the number of elements in the array, though it may be less (this is useful so that one does not have to create a new array if the array has trailing zeros). All but the first `n` terms of the array are ignored.

`S(a::RingElem)`

Try to coerce a into the base ring of the power series ring and then construct the power series $a + O(x^p)$.

`O(a::PowerSeries{T})`

The power series a should be of the form `x^n` for some nonnegative `n`. The function then returns $0 + O(x^n)$ with the same parent as the supplied power series.

If the power series 0 is passed to this function we throw a `DomainError()`.

Note that whilst we do not disallow it, we do not give any special meaning to expressions like $O(x - 1)$. Only the highest degree of the nonzero terms is observed.

Examples.

Here are some examples of power series constructors.

```
R, t = PolynomialRing(QQ, "t")
S, x = PowerSeriesRing(R, 30, "x")

a = x^3 + 2x + 1
b = (t^2 + 1)*x^2 + (t + 3)x + O(x^4)

c = S(a)
d = S([t + 1, t, R(1)], 3, 5)

g = S(1)
h = S(fmpz(2))
k = S()
```

6.3.2 Basic manipulation

`length{T <: RingElem}(a::PowerSeries{T})`

Return the length of the polynomial underlying the power series a, i.e. its degree plus one. Recall that power series in Nemo are stored in absolute format, not relative format, so the length of the underlying polynomial may be much bigger than the precision due to leading zeros.

`normalise{T <: RingElem}(a::PowerSeries{T}, len::Int)`

Return the normalised length of the polynomial representing the power series `a`, assuming that its unnormalised length is `len`. The polynomial representing `a` is normalised if it has either length 0 or its leading coefficient is nonzero. This function is mainly used internally because all user functions in Nemo normalise the internal polynomial representation of power series.

`coeff{T <: RingElem}(a::PowerSeries{T}, n::Int)`

Return the coefficient of the power series `a` with degree `n`. If $n < 0$ we throw a `DomainError()` and if n is greater than or equal to the power series precision we return 0.

`zero{T <: RingElem}(R::PowerSeriesRing{T})`

Return $0+O(x^p)$ where p is the maximum precision for the given power series ring and where x is its variable.

`one{T <: RingElem}(::Type{PowerSeries{T}})`

Return $1+O(x^p)$ where p is the maximum precision for the given power series ring and where x is its variable.

`gen{T <: RingElem}(::Type{PowerSeries{T}})`

Return $x + O(x^{p+1})$ where p is the maximum precision for the given power series ring and where x is its variable.

`iszero{T <: RingElem}(a::PowerSeries{T})`

Return `true` if a is arithmetically equal to 0 in the power series ring up to the precision of a.

`isone{T <: RingElem}(a::PowerSeries{T})`

Return `true` if a is arithmetically equal to 1 in the power series ring up to the precision of a.

`isgen{T <: RingElem}(a::PowerSeries{T})`

Return `true` if a is arithmetically equal to x in the power series ring up to the precision of a.

`isunit{T <: RingElem}(a::PowerSeries{T})`

Return `true` if a is an invertible element of the power series ring, i.e. if the lead term is degree zero with invertible coefficient. Otherwise, return `false`.

```
valuation{T <: RingElem}(a::PowerSeries{T})
```

Return the valuation of the power series `a` with respect to the generator of the power series ring. If the power series is zero to finite precision n, the valuation is defined to be n.

```
precision{T <: RingElem}(a::PowerSeries{T})
```

Return the (absolute) precision of the power series `a`.

```
max_precision{T <: RingElem}(R::PowerSeriesRing{T})
```

Return the maximum precision for the given power series ring. Recall that this is a relative precision.

```
deepcopy{T <: RingElem}(a::PowerSeries{T})
```

Create a new power series object which is precisely equal to the supplied power series, i.e. with the same coefficients and precision.

Examples.

Here are some examples of basic manipulations of power series.

```
R, t = PolynomialRing(QQ, "t")
S, x = PowerSeriesRing(R, 30, "x")

a = O(x^4)
b = (t^2 + 1)*x^2 + (t + 3)x + O(x^4)

c = gen(R)
d = zero(R)
f = one(R)

g = iszero(d)
h = isone(f)
k = isgen(c)
m = isunit(-1 + x + 2x^2)
n = valuation(a)
p = valuation(b)
q = deepcopy(a)
```

6.3.3 Unary operators

`-{T <: RingElem}(a::PowerSeries{T})`
Return $-a$.

Examples.

Here are some examples of unary operators.

```
R, t = PolynomialRing(QQ, "t")
S, x = PowerSeriesRing(R, 30, "x")

a = 2x + x^3
b = 1 + 2x + x^2 + O(x^3)

c = -a
d = -b
```

6.3.4 Binary operators

`+{T <: RingElem}(a::PowerSeries{T}, b::PowerSeries{T})`
Return $a + b$. If the two power series have differing absolute precisions, the result will be a power series with the lower precision.

`-{T <: RingElem}(a::PowerSeries{T}, b::PowerSeries{T})`

Return $a - b$. If the two power series have differing absolute precisions, the result will be a power series with the lower precision.

`*{T <: RingElem}(a::PowerSeries{T}, b::PowerSeries{T})`

Return ab. The return precision is equal to the least out of the sum of the precision of `a` and the valuation of `b` and the sum of the valuation of `a` and the precision of `b`.

Examples.

Here are some examples of binary operators.

```
R, t = PolynomialRing(QQ, "t")
S, x = PowerSeriesRing(R, 30, "x")

a = 2x + x^3
b = O(x^4)
c = 1 + x + 3x^2 + O(x^5)
d = x^2 + 3x^3 - x^4

f = a + b
g = a - c
h = b*c
j = a*c
m = a*d
```

6.3.5 Ad hoc binary operators

```
*{T <: RingElem}(a::Int, b::PowerSeries{T})
*{T <: RingElem}(a::fmpz, b::PowerSeries{T})
```

Return ab. The resulting power series will have the same precision as b.

```
*{T <: RingElem}(a::PowerSeries{T}, b::Int)
*{T <: RingElem}(a::PowerSeries{T}, b::fmpz)
```

Return ab. The resulting power series will have the same precision as a.

Examples.

Here are some examples of ad hoc binary operators.

```
R, t = PolynomialRing(QQ, "t")
S, x = PowerSeriesRing(R, 30, "x")

a = 2x + x^3
b = O(x^4)
c = 1 + x + 3x^2 + O(x^5)
d = x^2 + 3x^3 - x^4

f = 2a
g = fmpz(3)*b
h = c*2
j = d*fmpz(3)
```

6.3.6 Comparison

Note that we take the convention that $1 + x + O(x^4) == 1 + x + O(x^8)$.

```
==(T <: RingElem}(x::PowerSeries{T}, y::PowerSeries{T})
```

Return `true` if the two power series are the same up to the lesser of the precisions of the two series, otherwise return `false`.

Julia automatically defines a corresponding `!=` operator.

```
isequal(x::PowerSeries{T}, y::PowerSeries{T})
```

Returns `true` if the $x == y$, the precisions of x and y are the same and if each coefficient of x recursively compares equal to the corresponding coefficient of y according to `isequal`, otherwise return `false`.

Examples.

Here are some examples of comparison.

```
R, t = PolynomialRing(QQ, "t")
S, x = PowerSeriesRing(R, 30, "x")

a = 2x + x^3
b = O(x^3)
c = 1 + x + 3x^2 + O(x^5)
d = 3x^3 - x^4

a == 2x + x^3
b == d
c != d
isequal(b, d) == false
```

6.3.7 Ad hoc comparison

Note that we take the convention that $0 == 0 + O(x^4)$.

```
==(T <: RingElem}(x::PowerSeries{T}, y::Int)
==(T <: RingElem}(x::PowerSeries{T}, y::fmpz)
```

> Return `true` if the power series `x` is equal to `y` up to the precision it has. Note that this is always true if `x` has precision 0.

```
==(T <: RingElem}(x::Int, y::PowerSeries{T})
==(T <: RingElem}(x::fmpz, y::PowerSeries{T})
```

> Return `true` if the power series `y` is equal to `x` up to the precision it has. Note that this is always true if `y` has precision 0.

Julia automatically defines a corresponding `!=` operator corresponding to the above.

Examples.

Here are some examples of ad hoc comparison.

```
R, t = PolynomialRing(QQ, "t")
S, x = PowerSeriesRing(R, 30, "x")

a = 2x + x^3
b = O(x^0)
c = 1 + O(x^5)
d = S(3)

d == 3
c == fmpz(1)
fmpz(0) != a
2 == b
fmpz(1) == c
```

6.3.8 Powering

```
^{T <: RingElem}(a::PowerSeries{T}, b::Int)
```

Return a^b. The result will have precision equal to $b - 1$ times the valuation of a plus the precision of a. This is the same as if a were multiplied by itself b times.

Examples.

Here are some examples of powering.

```
R, t = PolynomialRing(QQ, "t")
S, x = PowerSeriesRing(R, 30, "x")

a = 2x + x^3
b = O(x^4)
c = 1 + x + 2x^2 + O(x^5)
d = 2x + x^3 + O(x^4)

f = a^12
g = b^12
h = c^12
k = d^12
```

6.3.9 Shifting

```
shift_left{T <: RingElem}(a::PowerSeries{T}, n::Int)
```

Return `a*x^n`, where `x` is the generator of the power series ring and `n` is a nonnegative integer. If `n` is negative we raise a `DomainError()`.

```
shift_right{T <: RingElem}(a::PowerSeries{T}, n::Int)
```

Return `a/x^n`, discarding the remainder, where `x` is the generator of the power series ring and `n` is a nonnegative integer. If `n` is negative we raise a `DomainError()`.

Examples.

Here are some examples of shifting.

```
R, t = PolynomialRing(QQ, "t")
S, x = PowerSeriesRing(R, 30, "x")

a = 2x + x^3
b = O(x^4)
c = 1 + x + 2x^2 + O(x^5)
d = 2x + x^3 + O(x^4)

f = shift_left(a, 2)
g = shift_left(b, 2)
h = shift_right(c, 1)
k = shift_right(d, 3)
```

6.3.10 Truncation

`truncate{T <: RingElem}(a::PowerSeries{T}, prec::Int)`

Return the power series `a` truncated to the given (absolute) precision. If `a` already has precision less than or equal to `prec` this results in no change. If `prec` is negative we throw a `DomainError()`.

Examples.

Here are some examples of truncation.

```
R, t = PolynomialRing(QQ, "t")
S, x = PowerSeriesRing(R, 30, "x")

a = 2x + x^3
b = O(x^4)
c = 1 + x + 2x^2 + O(x^5)
d = 2x + x^3 + O(x^4)

f = truncate(a, 3)
g = truncate(b, 2)
h = truncate(c, 7)
k = truncate(d, 5)
```

6.3.11 Exact division

`divexact{T <: RingElem}(x::PowerSeries{T}, y::PowerSeries{T})`

Return the quotient of x by y. This requires `x` to have valuation at least that of `y`. It also requires the first nonzero coefficient of `y` to be a unit in the ring `T`. If `y` is zero, a `DivideError()` is thrown.

The precision of the result is equal to the minimum of $p_x - v_y$ and $p_y - 2v_y + v_x$ where p_x, p_y are the precisions of `x` and `y` respectively and v_x, v_y are the corresponding valuations.

Examples.

Here are some examples of exact division.

```
R, t = PolynomialRing(QQ, "t")
S, x = PowerSeriesRing(R, 30, "x")

a = x + x^3
b = O(x^4)
c = 1 + x + 2x^2 + O(x^5)
d = x + x^3 + O(x^6)

f = divexact(a, d)
g = divexact(d, a)
h = divexact(b, c)
k = divexact(d, c)
```

6.3.12 Ad hoc exact division

```
divexact{T <: RingElem}(x::PowerSeries{T}, y::Int)
divexact{T <: RingElem}(x::PowerSeries{T}, y::fmpz)
divexact{T <: RingElem}(x::PowerSeries{T}, y::T)
divexact{T <: RingElem}(x::PowerSeries{T}, y::Integer)
```

Return the quotient of x by y. If `y` is zero, a `DivideError()` is thrown.

The precision of the result is equal to that of x.

Examples.

Here are some examples of ad hoc exact division.

```
R, t = PolynomialRing(QQ, "t")
S, x = PowerSeriesRing(R, 30, "x")

a = x + x^3
b = O(x^4)
c = 1 + x + 2x^2 + O(x^5)
d = x + x^3 + O(x^6)

f = divexact(a, 7)
g = divexact(b, fmpz(11))
h = divexact(c, fmpz(2))
k = divexact(d, 9)
```

6.3.13 Inversion

```
inv{T <: RingElem}(a::PowerSeries{T})
```

Return the power series inverse of a. A `DivideError()` is thrown if `a` is zero. An exception is thrown if `a` isn't invertible.

The precision of the result is equal to the precision of `a`.

Examples.

Here are some examples of inversion.

```
R, t = PolynomialRing(QQ, "t")
S, x = PowerSeriesRing(R, 30, "x")

a = 1 + x + 2x^2 + O(x^5)
b = S(-1)

c = inv(a)
d = inv(b)
```

6.3.14 Special functions

`exp{T <: RingElem}(a::PowerSeries{T})`

Compute the power series exponential of the given power series `a` to the same precision as `a`. The valuation of `a` must be nonzero.

Examples.

Here are some examples of special functions.

```
R, t = PolynomialRing(QQ, "t")
S, x = PowerSeriesRing(R, 30, "x")

a = exp(x + O(x^40))
b = divexact(x, exp(x + O(x^40)) - 1)
```

6.4 Generic matrices (Mat(R)): `Mat{T}`

In Nemo we provide generic matric spaces for $m \times n$ matrices over a ring R. Note that $m \times n$ matrices do not form a ring.

The type of a parent object for a matrix space is a `MatrixSpace`. Currently this belongs to the `Ring` type class to accomodate the special case when working with $n \times n$ matrices. However this will change in a later version of Nemo when matrix algebras are provided.

Matrix elements are of type `Mat` which belongs to the `MatElem` type class, which in turn belongs to `RingElem`. Again, this is temporary to accomodate the special case where matrices happen to be square, forming an actual mathematical ring.

There's no need to deal directly with these types, as we provide a function `MatrixSpace` for constructing the matrix space parent objects and various constructors to create matrix objects.

As for other generic types, the `Mat` and `MatrixSpace` types take as parameters the type `T` of the elements in the base ring over which the matrix space is constructed.

For example, if there wasn't a special Flint type for matrices, the type of a matrix space $Mat_{m,n}(\mathbb{Z})$ over the Flint integers would be denoted `MatrixSpace{fmpz}` and elements of the matrix space would have type `Mat{fmpz}`.

6.4.1 Constructors

We define the following function for constructing the parent object of a matrix space.

`MatrixSpace(R::Ring, r::Int, c::Int)`

Returns a parent object `S` corresponding to the space of $r \times c$ matrices over the ring `R`

.

We also provide various constructors for creating elements of a matrix space. Below we assume that S is a matrix space parent object, created for example by the following code:

```
R, t = PolynomialRing(QQ, "t")
S = MatrixSpace(R, 3, 3)
```

In the following we will also let r stand for the number of rows and c for the number of columns, e.g. $r = 3$ and $c = 3$ in the case of S as we just defined it. We let T stand for the type of the entries of the matrices.

```
S()
```

Create the $r \times c$ matrix with all entries zero.

```
S(a::Integer)
S(a::T)
```

Create the $r \times c$ diagonal matrix, with diagonal entries equal to a (viewed as an element of the base ring).

```
S(a::Mat{T})
```

Return a reference to the matrix `a`. No copy of the data is made by this function.

```
S(a::Array{T, 2})
```

Create an $r \times c$ given by the entries of the $r \times c$ Julia array a.

```
S(a::RingElem)
```

Try to coerce a into the base ring of the matrix space and then construct the diagonal $r \times c$ matrix with diagonal entries equal to a.

Examples.

Here are some examples of power series constructors.

```
R, t = PolynomialRing(QQ, "t")
S = MatrixSpace(R, 3, 3)

A = S()
B = S(12)
C = S([t + 1 t R(1); t^2 t t; R(-2) t + 2 t^2 + t + 1])
D = S(c)
F = S(R(11))
```

6.4.2 Basic manipulation

To access entries of a matrix we overload Julia's `getindex` and `setindex` methods. For example, if A is a `Mat{T}`as defined in the example above we have the following matrix notation.

```
A[i::Int, j::Int]
```

When assigned to, this allows the entry at row i and column j to be assigned to. When used in an expression, this allows the entry at row i and column j of the matrix to be used in an expression. Note that Julia arrays and Nemo matrices are 1-indexed, i.e. the top left entry is $A[1, 1]$.

```
rows{T <: RingElem}(a::Mat{T})
```

Return the number of rows of the given matrix.

```
cols{T <: RingElem}(a::Mat{T})
```

Return the number of rows of the given matrix.

```
zero{T <: RingElem}(R::MatrixSpace{T})
```

Return the 0 matrix in the matrix space, i.e. the matrix with all entries equal to 0 in the base ring.

```
one{T <: RingElem}(::Type{MatrixSpace{T}})
```

Return 1 diagonal matrix in the matrix space, i.e. the matrix with all diagonal entries equal to 1 in the base ring.

```
iszero{T <: RingElem}(a::Mat{T})
```

Return `true` if a is arithmetically equal to the 0 matrix.

```
isone{T <: RingElem}(a::Mat{T})
```

Return `true` if a is arithmetically equal to the 1 diagonal matrix.

```
deepcopy{T <: RingElem}(a::Mat{T})
```

Create a new matrix object which is equal to the supplied matrix, i.e. with the same entries.

Examples.

Here are some examples of basic manipulation of matrices.

```
R, t = PolynomialRing(QQ, "t")
S = MatrixSpace(R, 3, 3)

A = S([t + 1 t R(1); t^2 t t; R(-2) t + 2 t^2 + t + 1])
B = S([R(2) R(3) R(1); t t + 1 t + 2; R(-1) t^2 t^3])

C = zero(S)
D = one(S)

f = iszero(C)
g = isone(D)
h = A[1, 2]
B[1, 1] = R(3)
r = rows(B)
c = cols(B)
k = deepcopy(A)
```

6.4.3 Unary operators

`-{T <: RingElem}(a::Mat{T})`
Return $-a$, i.e. the matrix with all entries negated.

Examples.

Here is an example of unary operators.

```
R, t = PolynomialRing(QQ, "t")
S = MatrixSpace(R, 3, 3)

A = S([t + 1 t R(1); t^2 t t; R(-2) t + 2 t^2 + t + 1])

B = -A
```

6.4.4 Binary operators

`+{T <: RingElem}(a::Mat{T}, b::Mat{T})`
Return $a + b$. The matrices must have the same dimensions otherwise an exception is thrown.

`-{T <: RingElem}(a::Mat{T}, b::Mat{T})`

Return $a - b$. The matrices must have the same dimensions otherwise an exception is thrown.

`-{T <: RingElem}(a::Mat{T}, b::Mat{T})`

Return $a - b$. The matrices must have the same dimensions otherwise an exception is thrown.

`*{T <: RingElem}(a::Mat{T}, b::Mat{T})`

Return $a - b$. The number of columns of a must equal the number of rows of b, otherwise an exception is thrown.

Examples.

Here is an example of unary operators.

```
R, t = PolynomialRing(QQ, "t")
S = MatrixSpace(R, 3, 3)

A = S([t + 1 t R(1); t^2 t t; R(-2) t + 2 t^2 + t + 1])
B = S([R(2) R(3) R(1); t t + 1 t + 2; R(-1) t^2 t^3])

C = A + B
D = A - B
F = A*B
```

6.4.5 Ad hoc binary operators

```
+{T <: RingElem}(a::Mat{T}, b::Integer)
+{T <: RingElem}(a::Mat{T}, b::fmpz)
+{T <: RingElem}(a::Mat{T}, b::T)
```

Return the sum of a and the diagonal matrix with b for the diagonal entries.

```
+{T <: RingElem}(a::Integer, b::Mat{T})
+{T <: RingElem}(a::fmpz, b::Mat{T})
+{T <: RingElem}(a::T, b::Mat{T})
```

Return the sum of the diagonal matrix with a for the diagonal entries and the matrix b.

```
-{T <: RingElem}(a::Mat{T}, b::Integer)
-{T <: RingElem}(a::Mat{T}, b::fmpz)
-{T <: RingElem}(a::Mat{T}, b::T)
```

Return the difference of a and the diagonal matrix with b for the diagonal entries.

```
-{T <: RingElem}(a::Integer, b::Mat{T})
-{T <: RingElem}(a::fmpz, b::Mat{T})
-{T <: RingElem}(a::T, b::Mat{T})
```

Return the difference of the diagonal matrix with a for the diagonal entries and the matrix b.

```
*{T <: RingElem}(a::Mat{T}, b::Integer)
*{T <: RingElem}(a::Mat{T}, b::fmpz)
*{T <: RingElem}(a::Mat{T}, b::T)
```

Return the matrix whose entries are those of a multiplied by b.

```
*{T <: RingElem}(a::Integer, b::Mat{T})
*{T <: RingElem}(a::fmpz, b::Mat{T})
*{T <: RingElem}(a::T, b::Mat{T})
```

Return the matrix whose entries are a multiplied by the entries of b.

Examples.

Here are some examples of ad hoc binary operators.

```
R, t = PolynomialRing(QQ, "t")
S = MatrixSpace(R, 3, 3)

A = S([t + 1 t R(1); t^2 t t; R(-2) t + 2 t^2 + t + 1])

B = A + 12
C = fmpz(11) + A
D = A - (t + 1)
F = A*3
G = fmpz(7)*A
H = (t - 1)*A
```

6.4.6 Powering

```
^{T <: RingElem}(a::Mat{T}, n::Int)
```

Return the matrix a raised to the power of n, i.e. multiplied by itself n times. For generic matrices over a ring, binary exponentiation is used. If the exponent is 0 the 1 diagonal matrix is returned. Currently negative powers are not supported and a `DomainError()` is raised in this case. Note that the matrix must be square for this operation to be performed, otherwise an exception is raised.

Examples.

Here is an example of powering.

```
R, t = PolynomialRing(QQ, "t")
S = MatrixSpace(R, 3, 3)

A = S([t + 1 t R(1); t^2 t t; R(-2) t + 2 t^2 + t + 1])

B = A^12
```

6.4.7 Comparisons

```
==(T <: RingElem}(a::Mat{T}, b::Mat{T})
```

Return `true` if the two matrices are equal, i.e. if corresponding entries are equal, otherwise return `false`. The matrices must have the same dimensions, otherwise an exception is raised.

Examples.

Here are some examples of comparison.

```
R, t = PolynomialRing(QQ, "t")
S = MatrixSpace(R, 3, 3)

A = S([t + 1 t R(1); t^2 t t; R(-2) t + 2 t^2 + t + 1])
B = S([R(2) R(3) R(1); t t + 1 t + 2; R(-1) t^2 t^3])

A != B
A == deepcopy(A)
```

6.4.8 Ad hoc comparisons

```
==(T <: RingElem}(a::Mat{T}, b::Integer)
==(T <: RingElem}(a::Mat{T}, b::fmpz)
```

Return `true` if the matrix a is equal to the diagonal matrix of the same dimensions with b for the diagonal entries, otherwise return `false`.

```
==(T <: RingElem}(a::Integer, b::Mat{T})
==(T <: RingElem}(a::fmpz, b::Mat{T})
```

Return `true` if the diagonal matrix of the same dimensions as b, with a for the diagonal entries is equal to the matrix b, otherwise return `false`.

Examples.

Here are some examples of ad hoc comparison.

```
R, t = PolynomialRing(QQ, "t")
S = MatrixSpace(R, 3, 3)

A = S([t + 1 t R(1); t^2 t t; R(-2) t + 2 t^2 + t + 1])

A != 12
fmpz(11) != A
S(11) == 11
```

6.4.9 Ad hoc exact division

```
divexact{T <: RingElem}(a::Mat{T}, b::Integer)
divexact{T <: RingElem}(a::Mat{T}, b::fmpz)
divexact{T <: RingElem}(a::Mat{T}, b::T)
```

Return a/b, i.e. the matrix whose entries are those of a divided by b. It is assumed that each such division is exact.

Examples.

Here are some examples of ad hoc exact division.

```
R, t = PolynomialRing(QQ, "t")
S = MatrixSpace(R, 3, 3)

A = S([t + 1 t R(1); t^2 t t; R(-2) t + 2 t^2 + t + 1])

B = divexact(A, -1)
C = divexact(12*A, fmpz(3))
D = divexact((t^2 - 1)*A, t + 1)
```

6.4.10 Gram matrix

```
gram{T <: RingElem}(a::Mat{T})
```

Return the Gram matrix of a, i.e. if a is an $r \times c$ matrix return the $r \times r$ matrix whose entries i, j are the dot products of the i-th and j-th rows, respectively.

Examples.

Here is an example of computing the Gram matrix.

```
R, t = PolynomialRing(QQ, "t")
S = MatrixSpace(R, 3, 3)

A = S([t + 1 t R(1); t^2 t t; R(-2) t + 2 t^2 + t + 1])

B = gram(A)
```

6.4.11 Trace

`trace{T <: RingElem}(a::Mat{T})`

Return the trace of the matrix a, i.e. the sum of the diagonal elementst. We require the matrix to be square.

Examples.

Here is an example of computing the trace of a matrix.

```
R, t = PolynomialRing(QQ, "t")
S = MatrixSpace(R, 3, 3)

A = S([t + 1 t R(1); t^2 t t; R(-2) t + 2 t^2 + t + 1])

b = trace(A)
```

6.4.12 Content

`content{T <: RingElem}(a::Mat{T})`

Return the content of the matrix a, i.e. the greatest common divisor of all its entries, assuming it exists.

Examples.

Here is an example of computing the content of a matrix.

```
R, t = PolynomialRing(QQ, "t")
S = MatrixSpace(R, 3, 3)

A = S([t + 1 t R(1); t^2 t t; R(-2) t + 2 t^2 + t + 1])

b = content(A)
```

7 Generic fields

Nemo provides various constructions of fields.

Nemo defines the abstract type `Field` like so:

```
abstract Field <: Ring
```

This means fields can be used anywhere that a more general ring can be used in Nemo.

Division in a field can be performed with `divexact`. Of course all divisions in a field are exact, except division by zero which will raise a `DivideError()`.

The `//` operator doubles as a division operator when passed operands in a field. It is an alias for `divexact`.

7.1 Fraction fields Frac(R): `Fraction{T}`

We allow the construction of a fraction field over any ring in Nemo. Not all such constructions make sense. Typically it only makes sense to construct the fraction field of an integral domain. However, so long as zero divisors are avoided there is nothing stopping one constructing the fraction field of a more general Euclidean ring in Nemo.

Specifically, a ring must provide a `gcd` function before we can take its fraction field. This is required for canonicalisation.

The type of an element of a fraction field in Nemo is `Fraction` which belongs to the type class `FractionElem` which in turn belongs to `FieldElem`. The type of a fraction field parent object is `FractionField` which belongs to `Field`.

The user need never deal directly with these types, as we provide convenient functions for constructing objects of those types.

The types of `FractionField` parent objects and `Fraction` elements are parameterised by the type `T` of elements of the base ring of the fraction field.

The generic `Fraction` objects contain a numerator and denominator, both of type `T`, and a pointer to the parent fraction field object.

The fraction field parent objects can be used to construct fractions in the given fraction field.

The fractions in Nemo are canonicalised, which means that we divide both numerator and denominator by their greatest common divisor. We also divide by `canonical_unit(d)` where `d` is the denominator when needed, e.g. when printing fractions or taking their numerator or denominator, etc. We also use this to easily check equality of fractions.

The `canonical_unit` function must have the properties

```
canonical_unit(u) == u
canonical_unit(a*b) == canonical_unit(a)*canonical_unit(b)
```

for all nonzero values a and b in the base ring and u any invertible element in the base ring. Moreover, `canonical_unit` must always return an invertible element in the base ring for nonzero elements of the ring.

We overload the `//` operator in Nemo for rings whose fraction field can be taken (e.g. `fmpz` and `Poly{T}`), so that elements of the fraction field of a ring can be constructed directly, without first constructing the type.

For example if we have

```
R, x = PolynomialRing(ZZ, "x")

s = (x + 1)//(x^2 + 1)
```

then the value s belongs to the fraction field of R automatically, even though this field has not been constructed yet.

In Nemo, we consider the fraction field of a field to be itself.

7.1.1 Constructors

We provide a convenient function for constructing a fraction field of a ring in Nemo.

```
FractionField(R::Ring)
```

Construct the fraction field of the given ring.

There are also numerous constructors for generating elements in the fraction field. In the constructor below, we assume S is the fraction field of a ring, e.g. as constructed by the following:

```
R, x = PolynomialRing(ZZ, "x")
S = FractionField(R)
```

We assume T represents the type of elements of the base ring.

```
S(a::T)
```

Construct the canonicalised fraction `a//1` in the fraction field.

```
S(a::Integer)
S(a::T)
```

Return the value $a//1$ in the fraction field.

```
S(a::Integer, b::Integer)
S(a::T, b::T)
S(a::T, b::Integer)
S(a::Integer, b::T)
```

Construct the canonicalised fraction `a//b` in the fraction field. If $b = 0$ we throw a `DivideError()` and an exception is raised in general if b is not invertible.

```
S()
```

Return the value $0/1$ in the fraction field.

```
S(a::Fraction{T})
```

Return a reference to the value a. No copy of the data is made.

```
S(b::RingElem)
```

Coerce b into the base ring of the fraction field and construct the fraction $b//1$.

We also provide a simpler way of constructing elements of a fraction field without needing to first construct the type of the fraction field.

```
//(x::T, y::T)
//(x::T, y::Integer)
//(x::Integer, y::T)
```

Return the canonicalised value $x//y$ in the fraction field. In the cases where one argument is an integer, it is first coerced into the base ring. If $y = 0$ we throw a `DivideError()` and an exception is raised in general if y is not invertible.

Examples.

Here are some examples of constructors for fraction fields.

```
S, x = PolynomialRing(ZZ, "x")
T = FractionField(S)

b = T(3)
c = T(fmpz(7))
d = T(x + 2)
f = T(d)
g = T()
h = T(x + 1, x + 2)

k = (x + 3)//(x^2 + 2)
```

7.1.2 Basic manipulation

`num{T <: RingElem}(a::Fraction{T})`

Return the numerator of the fraction a.

`den{T <: RingElem}(a::Fraction{T})`

Return the denominator of the fraction a.

`zero{T <: RingElem}(R::FractionField{T})`

Return the additive identity of the fraction field of the ring T.

`one{T <: RingElem}(R::Type{FractionField{T}})`

Return the multiplicative identity of the fraction field of the ring T.

`iszero{T <: RingElem}(a::Fraction{T})`

Return `true` if the fraction a is the additive identity in the fraction field, otherwise return `false`.

`isone{T <: RingElem}(a::Fraction{T})`

Return `true` if the fraction a is multiplicative identity in the fraction field, otherwise return `false`.

`isunit{T <: RingElem}(a::Fraction{T})`

Return `true` if the fraction a is invertible, i.e. is not zero, otherwise return `false`.

`canonical_unit{T}(a::Fraction{T})`

Used for canonicalising fractions. This function simply returns a.

```
deepcopy{T}(a::Fraction{T})
```

Creates a new fraction object arithmetically equal to a.

Examples.

Here are some examples of basic manipulations for fraction fields.

```
S, x = PolynomialRing(ZZ, "x")

a = den((x + 1)//(-x^2 + 1))

c = zero(S)
d = one(S)
f = canonical_unit((x + 1)//(-x^2 + 1))
g = isunit((x + 1)//(-x^2 + 1))

h = iszero(c)
d = isone(d)
f = deepcopy(d)
```

7.1.3 Unary operators

```
-{T <: RingElem}(a::Fraction{T})
```

Return $-a$.

Examples.

Here is an example of a unary operator.

```
S, x = PolynomialRing(ZZ, "x")

a = -((x + 1)//(-x^2 + 1))
```

7.1.4 Binary operators and functions

The following binary operators are available for elements of fraction fields. All of the functions canonicalise their outputs.

```
+{T <: RingElem}(a::Fraction{T}, b::Fraction{T})
```

Return $a + b$.

```
-{T <: RingElem}(a::Fraction{T}, b::Fraction{T})
```

Return $a - b$.

```
*{T <: RingElem}(a::Fraction{T}, b::Fraction{T})
```

Return $a * b$.

Examples.
Here are some examples of binary operators.

```
S, x = PolynomialRing(ZZ, "x")

a = -(x + 3)//(x + 1) + (2x + 3)//(x^2 + 4)
b = (x + 1)//(-x^2 + 1) - x//(2x + 1)
c = ((x^2 + 3x)//(5x))*((x + 1)//(2x^2 + 2))
```

7.1.5 Ad hoc binary operators

The following binary operators are more efficient than simply coercing both arguments into the fraction field. All functions canonicalise their outputs.

```
+{T <: RingElem}(a::Fraction{T}, b::Integer)
+{T <: RingElem}(a::Fraction{T}, b::T)
```

Return $a + b/1$.

```
+{T <: RingElem}(a::Integer, b::Fraction{T})
+{T <: RingElem}(a::T, b::Fraction{T})
```

Return $a/1 + b$.

```
-{T <: RingElem}(a::Fraction{T}, b::Integer)
-{T <: RingElem}(a::Fraction{T}, b::T)
```

Return $a - b/1$.

```
-{T <: RingElem}(a::Integer, b::Fraction{T})
-{T <: RingElem}(a::T, b::Fraction{T})
```

Return $a/1 - b$.

```
*{T <: RingElem}(a::Fraction{T}, b::Integer)
*{T <: RingElem}(a::Fraction{T}, b::T)
```

Return $a * b/1$.

```
*{T <: RingElem}(a::Integer, b::Fraction{T})
*{T <: RingElem}(a::T, b::Fraction{T})
```

Return $a/1 * b$.

Examples.
Here are some examples of ad hoc binary operators.

```
S, x = PolynomialRing(ZZ, "x")

a = (x + 1)//(-x^2 + 1)

c = a + 2
d = 3 - a
e = b*(x + 1)
```

7.1.6 Comparison operators

`==(T)(x::Fraction{T}, y::Fraction{T})`
Return `true` if the fraction x equals the fraction y, else return `false`.

`isequal(T)(x::Fraction{T}, y::Fraction{T})`

Return `true` if the numerators of `x` times the denominator of `y` compares equal recursively according to `isequal` with the denominator of `x` times the numerator of `y`, else return `false`.

Examples.
Here are some examples of comparisons.

```
S, x = PolynomialRing(ZZ, "x")

a = -(ZZ(4)/ZZ(6))
b = -((x + 1)/(-x^2 + 1))

a == -ZZ(2)/ZZ(3)
b == 1/(x - 1)
```

7.1.7 Ad hoc comparison

The following comparison operators are faster than first coercing both operands into the fraction field.

`==(T)(x::Fraction{T}, y::T)`

Return `true` if the fraction x equals the fraction $y/1$, else return `false`.

`==(T)(x::T, y::Fraction{T})`

Return `true` if the fraction y equals the fraction $x/1$, else return `false`.

`==(T <: RingElem)(x::Fraction{T}, y::Integer)`

Return `true` if the numerator of the fraction x is equal to y in the ring T and the denominator is 1, otherwise return `false`.

`==(T <: RingElem)(x::Integer, y::Fraction{T})`

Return `true` if the numerator of the fraction y is equal to x in the ring T and the denominator is 1, otherwise return `false`.

Examples.
Here are some examples of ad hoc comparisons.

```
S, x = PolynomialRing(ZZ, "x")

a = 1//(x - 1)

1//a == x - 1
1 == one(S)
```

7.1.8 Powering

```
^{T <: RingElem}(a::Fraction{T}, b::Int)
```
Return a^b. The exponent may be negative.

Examples.

Here are some examples of powering.

```
S, x = PolynomialRing(ZZ, "x")

a = (x + 1)//(-x^2 + 1)

c = a^(-12)
```

7.1.9 Inversion

```
inv{T <: RingElem}(a::Fraction{T})
```
Return $1/a$.

Examples.

Here is an example of inversion.

```
S, x = PolynomialRing(ZZ, "x")

a = (x + 1)//(-x^2 + 1)

b = inv(a)
```

7.1.10 Exact division

```
//{T <: RingElem}(a::Fraction{T}, b::Fraction{T})
```
Return the $a * inv(b)$.

```
divexact{T <: RingElem}(a::Fraction{T}, b::Fraction{T})
```

An alias for `a//b`.

Examples.

Here are an example of exact division.

```
S, x = PolynomialRing(ZZ, "x")

a = (x + 1)//(-x^2 + 1)
b = -x//(2x + 1)

c = a//c
```

7.1.11 Ad hoc exact division

```
divexact{T <: RingElem}(a::Fraction{T}, b::Integer)
divexact{T <: RingElem}(a::Fraction{T}, b::T)
```

Return $a/(b/1)$.

```
divexact{T <: RingElem}(a::Integer, b::Fraction{T})
divexact{T <: RingElem}(a::T, b::Fraction{T})
```

Return $(a/1)/b$.

Examples.

Here are some examples of ad hoc binary operators.

```
S, x = PolynomialRing(ZZ, "x")

a = (x^2)//(2x + 1)
b = (x + 1)//(-x^2 + 1)

f = a//5
g = (x + 1)//b
h = 3//b
```

7.1.12 GCD

```
gcd{T <: RingElem}(a::Fraction{T}, b::Fraction{T})
```

Return c/d where c is the GCD of the numerators of a and b and d is the GCD of their denominators.

Examples.

Here are some examples of binary operators and functions.

```
S, x = PolynomialRing(ZZ, "x")

a = -x//(2x + 1)

f = gcd(a, (x + 1)/(x - 1))
```

8 Flint rings

Flint provides implementations of various explicit rings. These are used instead of generic implementations where available.

We only describe the details that differ from the generic implementations above.

8.1 General Flint functions

`flint_cleanup()`

Free any internal caches used by Flint, Arb and Antic. Examples of cached data include `mpz` integers used internally to speed up allocation of bignums, prime numbers, and high-precision values of mathematical constants. Normally, these caches should not take up too much memory, but there may be extreme circumstances where the user needs to free them manually. This function is safe to call at any time.

`flint_set_num_threads(n::Int)`

Set the number of threads that Flint may use internally (the default value is 1). Most functions are currently unaffected by this setting.

Examples.

Here are some examples of the above functions.

```
flint_cleanup()

flint_set_num_threads(2)

flint_set_num_threads(1)
```

8.2 Flint integers ($\mathbb{Z}$): `fmpz`

Multi-precision integers can be created in Nemo using `ZZ`. For example, to create the integer 123 one can write:

```
ZZ(123)
```

By default, `ZZ` creates objects of the Flint `fmpz` type. This is achieved internally by setting `ZZ = FlintIntegerRing()`, which makes `ZZ` the unique object of type `FlintIntegerRing`.

One can also create Flint integers using the `fmpz` constructor directly:

```
fmpz(123)
```

In Nemo library code we use `fmpz` rather than `ZZ` since the latter is defined only for the convenience of the user and its definition may be changed by the user. It is also slightly more efficient to call `fmpz` directly, since `ZZ` is a global variable.

The type of an `fmpz` object is `fmpz`. Thus the following returns `Nemo.fmpz`:

```
typeof(fmpz(123))
```

An object of this type corresponds directly to the Flint `fmpz` C type.

The parent of an `fmpz` object is the unique object of type `FlintIntegerRing`. Thus the following returns "Integer Ring":

```
parent(fmpz(123))
```

The `FlintIntegerRing` type belongs to the `Ring` type class and the `fmpz` type belongs to the `RingElem` type class. Thus both of the following return true:

```
fmpz <: RingElem
FlintIntegerRing <: Ring
```

8.2.1 Constructors

Nemo provides various constructors for the `fmpz` type. Space is automatically recovered when the created object goes out of scope.

`fmpz()`

Creates an initialised `fmpz`. Its value will be 0.

`fmpz(n::Int)`

Create an `fmpz` with the given signed machine word value n.

`fmpz(n::BigInt)`

Create an `fmpz` with the given Julia `BigInt` value n.

`fmpz(n::Integer)`

Create an `fmpz` with the given value n, where n has any type belonging to the Julia `Integer` type class.

```
fmpz(n::Float16)
fmpz(n::Float32)
fmpz(n::Float64)
fmpz(n::BigFloat)
```

If n is an exact integer, return the `fmpz` with the value n, otherwise raise an `InexactError`.

`fmpz(s::String)`

Create an `fmpz` whose value is represented (in decimal) by the given string s. The string should not contain any whitespace, and can optionally begin with a $-$ sign.

`fmpz(x::fmpz)`

Returns a reference to x. No copy of the data is made.

`deepcopy(x::fmpz)`

Create a new `fmpz` whose value is arithmetically equal to that of x.

Examples.

Here are some examples of the above constructors.

```
a = fmpz(-123)
b = fmpz(12.0)
c = fmpz("-12345678765456783765456789000000000000000000000000000")
d = fmpz(BigFloat(10)^100)
e = fmpz(c)
f = deepcopy(c)
```

8.2.2 Conversions

It is possible to convert `fmpz` values to a variety of other types. This is achieved by overloading Julia's `convert` function.

```
Int(x::fmpz)
UInt(x::fmpz)
BigInt(x::fmpz)
Float64(x::fmpz)
Float32(x::fmpz)
Float16(x::fmpz)
BigFloat(x::fmpz)
```

Convert the `fmpz` x to the given type.

Examples.

Here are some examples of the above constructors.

```
a = fmpz(-123)
b = fmpz(12)

c = Int(a)
d = UInt(b)
f = Float64(a)
g = BigFloat(a)
```

8.2.3 Basic manipulation

Various functions are provided to do basic manipulations of `fmpz`'s.

```
one(::FlintIntegerRing)
```

Create the multiplicative identity element in the ring of integers.

```
zero(::FlintIntegerRing)
```

Create the additive identity element in the ring of integers.

```
isone(a::fmpz)
```

Return `true` if a is the multiplicative identity 1, otherwise return `false`.

```
iszero(a::fmpz)
```

Return `true` if a is the additive identity 0, otherwise return `false`.

```
isunit(a::fmpz)
```

Return `true` if $a = 1$ or $a = -1$, otherwise return `false`.

```
sign(a::fmpz)
```

Return either -1, 0 or 1 depending on whether the sign of a is negative, zero or positive, respectively. The returned value is of type `Int`.

```
fits(::Type{Int}, a::fmpz)
fits(::Type{Uint}, a::fmpz)
```

Return `true` if the value a will fit into a variable of the given type. Otherwise return `false`.

```
size(a::fmpz)
```

Return the number of machine words that make up the bignum a. The returned value has type `Int`.

```
canonical_unit(a::fmpz)
```

This is used for canonicalising fractions. The function simply returns -1 if $a < 0$, otherwise it returns 1.

```
num(a::fmpz)
```

For convenience when working with rationals. Simply returns the value of `a`.

```
den(a::fmpz)
```

For convenience when working with rationals. Always returns `fmpz(1)`.

Examples.

Here are some examples of the above basic manipulations.

```
a = one(FlintIntegerRing()) # by default one(ZZ) is the same thing
b = zero(FlintIntegerRing()) # by default zero(ZZ) is the same thing

if sign(a) < 0
   println("Negative")
end

if fits(Int, a)
   println("Fits into an Int")
end

s = size(a)

t = canonical_unit(fmpz(-12))

u = isunit(fmpz(-1))
v = iszero(b)
w = isone(a)
```

8.2.4 Binary operators

The following standard binary operators are provided for `fmpz`. Note that `$` is Julia's xor operator.

```
+(a::fmpz, b::fmpz)
-(a::fmpz, b::fmpz)
*(a::fmpz, b::fmpz)
%(a::fmpz, b::fmpz)
&(a::fmpz, b::fmpz)
|(a::fmpz, b::fmpz)
$(a::fmpz, b::fmpz)
```

Note that `%` follows the C (and Julia) convention of rounding the quotient towards zero. If $b = 0$ is passed to `%`, a `DivideError()` is thrown.

Julia automatically provides all of the combined assignment operators `+=, *=, &=` etc.

Here are some examples of binary operators.

```
a = fmpz(12)
b = fmpz(26)

c = a + b
d = a - b
```

8.2.5 Integer division

Various kinds of integer division are provided.

```
fdiv(a::fmpz, b::fmpz)
cdiv(a::fmpz, b::fmpz)
tdiv(a::fmpz, b::fmpz)
div(a::fmpz, b::fmpz)
```

> Integer division with rounding towards $-\infty$, $+\infty$ and 0 respectively. Note `div` is a synonym for `tdiv`. If $b = 0$ is passed to any of these functions, a `DivideError()` is thrown.

Here are some examples of integer division.

```
a = fmpz(12)
b = fmpz(5)

c = cdiv(a, b)
d = fdiv(a, b)
```

8.2.6 Remainder

```
mod(a::fmpz, b::fmpz)
mod(x::fmpz, c::Int)
rem(a::fmpz, b::fmpz)
rem(x::fmpz, c::Int)
```

Integer remainder after division with rounding towards $-\infty$ and 0 respectively. Note that `%` is a synonym for `rem`. If $b = 0$ is passed to any of these functions, a `DivideError()` is thrown.

Here are some examples of the remainder functions.

```
a = fmpz(12)
b = fmpz(5)

c = mod(a, b)
d = rem(a, 3)
```

8.2.7 Exact division

```
divexact(a::fmpz, b::fmpz)
divexact(x::fmpz, c::Int)
```

Exact integer division. This is more efficient than the above division functions, but will only return a meaningful result if the division is exact, i.e. if b divides a. If $b = 0$ is passed to this function, a `DivideError()` is thrown.

Here is an example of exact division.

```
a = fmpz(12)
b = fmpz(6)

c = divexact(a, b)
c = divexact(a, 2)
```

8.2.8 GCD and LCM

```
gcd(a::fmpz, b::fmpz)
```

Returns the greatest common divisor of a and b. For convenience, $\gcd(a, 0) = a$ and $\gcd(0, b) = b$. The returned value is always nonnegative.

```
lcm(a::fmpz, b::fmpz)
```

Returns the lowest common multiple of a and b, i.e. $ab/\gcd(a, b)$ if $a, b > 0$. For convenience, $\operatorname{lcm}(a, 0) = 0$ and $\operatorname{lcm}(0, b) = 0$. The returned value is always nonnegative.

Here are some examples of GCD and LCM.

```
a = fmpz(12)
b = fmpz(15)

c = gcd(a, b)
d = lcm(a, b)
```

8.2.9 Integer logarithms

```
flog(a::fmpz, b::fmpz)
flog(a::fmpz, b::Int)
```

Returns $\log_b(a)$ rounded towards 0. Both a and b are required to be positive, otherwise a `DomainError()` exception is thrown.

```
clog(a::fmpz, b::fmpz)
clog(a::fmpz, b::Int)
```

Returns $\log_b(a)$ rounded towards ∞. Both a and b are required to be positive, otherwise a `DomainError()` exception is thrown.

Examples.

Here are some examples of integer logarithms.

```
a = fmpz(12)
b = fmpz(3)

c = clog(a, b)
d = flog(a, 2)
```

8.2.10 Ad hoc operators

Since the Julia types `Int` and `Uint` cannot be made members of the `Ring` class, we must define some ad hoc operators, so that values of type `fmpz` can be more easily combined with `Int` and `Uint` values.

```
+(x::fmpz, c::Integer)
+(x::fmpz, c::Int)
+(x::Integer, c::fmpz)
+(x::Int, c::fmpz)
-(x::fmpz, c::Integer)
-(x::fmpz, c::Int)
-(x::Integer c::fmpz)
-(x::Int, c::fmpz)
*(x::fmpz, c::Integer)
*(x::fmpz, c::Int)
*(x::Integer, c::fmpz)
*(x::Int, c::fmpz)
%(x::fmpz, c::Int)
```

Here are some examples of the ad hoc operators.

```
a = fmpz(-12)

b = 3 + a
c = a + 3
d = a - 3
e = 5 - a
f = a % 7
```

8.2.11 Ad hoc division

We also define the following, which have the same semantics as the corresponding functions above.

```
tdiv(x::fmpz, c::Int)
fdiv(x::fmpz, c::Int)
cdiv(x::fmpz, c::Int)
div(x::fmpz, c::Int)
```

Here are some examples of ad hoc division.

```
a = fmpz(-12)

b = tdiv(a, 5)
c = cdiv(a, 3)
```

8.2.12 Binary shifting

We also define the following, which have the same semantics as the corresponding functions above. There are also shift operators, which require the shift to be of type `Int`.

```
<<(x::fmpz, c::Int)
```

Return $2^c x$.

```
fdivpow2(x::fmpz, c::Int)
cdivpow2(x::fmpz, c::Int)
tdivpow2(x::fmpz, c::Int)
>>(x::fmpz, c::Int)
```

Return $x/2^c$ with rounding towards $-\infty$, $+\infty$ and 0 respectively. Note that `>>` is a synonym for `fdivpow2`.

Here are some examples of binary shifting.

```
a = fmpz(-12)

b = a << 3
c = a >> 1
d = fdivpow2(a, 2)
```

8.2.13 Powering

There is also the caret operator, which is used for powering in Julia (and Nemo), in line with the mathematical convention.

```
^(x::fmpz, a::Uint)
^(x::fmpz, a::Int)
```

Return x^a. Requires $a \geq 0$.

Examples.

Here is an example of the caret operator.

```
a = fmpz(-12)

b = a^10
```

8.2.14 Comparison operators and functions

The following standard comparison operators are provided for `fmpz`.

```
==(a::fmpz, b::fmpz)
>(a::fmpz, b::fmpz)
<(a::fmpz, b::fmpz)
<=(a::fmpz, b::fmpz)
>=(a::fmpz, b::fmpz)
```

Julia automatically provides a `!=` operator.

Note that all these comparisons yield a result of type `Bool`.

There are also the following functions.

```
isequal(x::fmpz, y::fmpz)
```

For `fmpz` this is just an alias for the `==` operator.

```
cmp(x::fmpz, y::fmpz)
```

Return -1 if $x < y$, 1 if $x > y$ and 0 if $x = y$.

```
cmpabs(x::fmpz, y::fmpz)
```

Return -1 if $|x| < |y|$, 1 if $|x| > |y|$ and 0 if $|x| = |y|$, where $|x|$ is the absolute value of x.

Examples.

Here are some examples of the comparison operators and functions.

```
a = fmpz(-12)
b = fmpz(5)

if a < b
   println("a < b")
end

if cmpabs(a, b) == 0
   println("The absolute values of a and b are equal")
end
```

8.2.15 Ad hoc comparison operators

We overload the comparison operators to more easily deal with values of type `Int`.

```
==(a::fmpz, b::Int)
==(a::Int, b::fmpz)
>(a::fmpz, b::Int)
>(a::Int, b::fmpz)
<(a::fmpz, b::Int)
<(a::Int, b::fmpz)
<=(a::fmpz, b::Int)
<=(a::Int, b::fmpz)
>=(a::fmpz, b::Int)
>=(a::Int, b::fmpz)
```

Examples.

Here are some examples of the ad hoc comparison operators.

```
a = fmpz(-12)

if a < 7
   println("a < 7")
end

if a == -12
   println("a == -12")
end
```

8.2.16 Unary operators

`-(x::fmpz)`

Return $-x$.

`~(x::fmpz)`

Return the bignum corresponding to logical complement of the twos complement representation of x, namely $-x - 1$.

Here are some examples of the unary operators.

```
a = -fmpz(12)
b = ~fmpz(-5)
```

8.2.17 Absolute value

`abs(x::fmpz)`

Return the absolute value of x, i.e. x if $x >= 0$ and $-x$ otherwise.

Examples.

Here is an example of the absolute value function.

```
a = fmpz(-12)

b = abs(a)
```

8.2.18 Division with remainder

```
fdivrem(a::fmpz, b::fmpz)
tdivrem(a::fmpz, b::fmpz)
divrem(a::fmpz, b::fmpz)
```

Return a tuple (q, r) consisting of the quotient and remainder after division of x by y, with rounding towards $-\infty$ and 0 respectively. Note that `divrem` is a synonym for `tdivrem`. If $b = 0$ is passed to any of these functions a `DivideError()` is thrown.

Examples.

Here are some examples of the division with remainder functions.

```
q, r = fdivrem(fmpz(12), fmpz(5))
q, r = tdivrem(fmpz(12), fmpz(5))
```

8.2.19 Roots

```
isqrt(x::fmpz)
```

Return the square root of x, rounded down to the nearest integer. If $x < 0$ a `DomainError()` is thrown.

```
isqrtrem(x::fmpz)
```

Return a tuple (s, r) consisting of the square root of x, rounded down to the nearest integer and the remainder, i.e. $r = x - s^2$. If $x < 0$ a `DomainError()` is thrown.

```
root(x::fmpz, n::Int)
```

Return the n-th root of x, rounded down to the nearest integer. If $x < 0$ and n is even, a `DomainError()` is thrown. If $n \leq 0$ a `DomainError()` is thrown.

Examples.

Here are some examples of the root taking functions.

```
s = isqrt(fmpz(12))
s, r = isqrtrem(fmpz(12))
r = root(fmpz(1000), 3)
```

8.2.20 Extended GCD

`gcdx(a::fmpz, b::fmpz)`

Return a tuple (g, s, t) consisting of the greatest common divisor of a and b and values s and t such that $g = as + bt$, with $-b \leq s \leq b$ and $-a \leq t \leq a$.

`gcdinv(a::fmpz, b::fmpz)`

Return a tuple (g, s) consisting of the greatest common divisor of a and b and a value $0 \leq s < b$ such that there exists a value t such that $g = as + bt$. If $a < 0$ or $b < a$, a `DomainError()` is thrown.

Examples.

Here are some examples of the extended GCD functions.

```
g, s, t = gcdx(fmpz(12), fmpz(5))
g, s = gcdinv(fmpz(5), fmpz(12))
```

8.2.21 Bit twiddling

Various functions are provided to deal with bignums on a binary bit level.

`popcount(x::fmpz)`

Return the number of binary ones in the binary representation of x. If $x < 0$ the function will return 0. The result that is returned is of type `Int`.

`prevpow2(x::fmpz)`

Return the largest power of 2 which does not exceed x. If $x < 0$ the result is set to `-prevpow2(-x)`.

`nextpow2(x::fmpz)`

Return the smallest power of 2 which is not less than x. If $x < 0$ the result is set to `-nextpow2(-x)`.

`trailing_zeros(x::fmpz)`

Return the number of trailing zeros in the binary representation of the absolute value of x.

`clrbit!(x::fmpz, n::Int)`

Set bit n in the binary representation of the absolute value of x to 0. Bits are numbered starting with $n = 0$ for the least significant bit. N.B. this function does not return the result, but alters the value of x. This is an unsafe operation which will change the value of all variables which contain a reference to x.

`setbit!(x::fmpz, n::Int)`

Set bit n in the binary representation of the absolute value of x to 1. Bits are numbered starting with $n = 0$ for the least significant bit. N.B. this function does not return the result, but alters the value of x. This is an unsafe operation which will change the value of all variables which contain a reference to x.

```
combit!(x::fmpz, n::Int)
```

Complement bit n in the binary representation of the absolute value of x. Bits are numbered starting with $n = 0$ for the least significant bit. N.B. this function does not return the result, but alters the value of x. This is an unsafe operation which will change the value of all variables which contain a reference to x.

Examples.

Here are some examples of the bit twiddling functions.

```
a = fmpz(12)

p = popcount(a)
b = nextpow2(a)
combit!(a, 2)
```

8.2.22 Alternative bases

```
base(n::fmpz, b::Int)
```

Return a unicode string giving the representation of n in the given base b. Valid bases are $2 \leq b \leq 62$. If an invalid base is applied an error is raised.

```
bin(n::fmpz)
```

Return a unicode string giving the representation of n in binary, i.e. base 2.

```
oct(n::fmpz)
```

Return a unicode string giving the representation of n in octal, i.e. base 8.

```
dec(n::fmpz)
```

Return a unicode string giving the representation of n in decimal, i.e. base 10.

```
hex(n::fmpz)
```

Return a unicode string giving the representation of n in hexadecimal, i.e. base 16.

```
ndigits(n::fmpz, b::Integer = 10)
```

Return an `Int` representing the number of digits required to represent the absolute value of n as a string in base b. The default is $b = 10$. Note that 0 always requires one digit as a string.

```
nbits(n::fmpz)
```

Return an `Int` representing the number of bits required to represent the absolute value of n in binary. We take the convention that 0 requires 0 binary bits.

Examples.

Here are some examples of the functions for alternative bases.

```
a = fmpz(12)

s1 = bin(a)
s2 = base(a, 13)
n1 = nbits(a)
n2 = ndigits(a, 3)
```

8.2.23 String I/O

```
string(x::fmpz)
```

Return a unicode string representation of the value of x in decimal, including the sign if $x < 0$.

Examples.

Here is an example of string I/O.

```
a = fmpz(12)

string(a)
```

8.2.24 Modular arithmetic

```
mod(x::fmpz, y::fmpz)
```

Return the Euclidean remainder of a by b. If $m = 0$ a `DivideError()` is thrown.

```
powmod(x::fmpz, p::Int, m::fmpz)
powmod(x::fmpz, p::fmpz, m::fmpz)
```

Return $x^p \pmod{m}$. If $m \leq 0$ a `DomainError()` is thrown.

```
invmod(x::fmpz, m::fmpz)
```

Return $x^{-1} \pmod{m}$. If $m \leq 0$ a `DomainError()` is thrown. If an impossible inverse is encountered, an exception is thrown.

```
gcdinv(x::fmpz, m::fmpz)
```

Returns the pair `g, xinv` where g is the greatest common divisor of x and m and `xinv` is $x^{-1} \pmod{m}$. If $m \leq 0$ a `DomainError()` is thrown.

```
sqrtmod(x::fmpz, m::fmpz)
```

Return the square root of x modulo m. If $m \leq 0$ a `DomainError()` is thrown. If x is not a square modulo m, an exception is thrown. Requires m to be prime. This condition is not checked and an infinite loop may result if m is not prime.

```
crt(r1::fmpz, m1::fmpz, r2::fmpz, m2::fmpz, sign = false)
crt(r1::fmpz, m1::fmpz, r2::Int, m2::Int, sign = false)
```

Return a value r such that $r \equiv r1 \pmod{m1}$ and $r \equiv r2 \pmod{m2}$. If `sign = true` the value will be in the range `-m1*m2/2 < r <= m1*m2/2`. If `sign = false` the value will be in the range `0 <= r < m1*m2`.

Examples.

Here are some examples of modular arithmetic.

```
a = powmod(fmpz(12), fmpz(110), fmpz(13))
a = powmod(fmpz(12), 110, fmpz(13))
b = invmod(fmpz(12), fmpz(13))
c = sqrtmod(fmpz(12), fmpz(13))
d = crt(fmpz(5), fmpz(13), fmpz(7), fmpz(37), true)
```

8.2.25 Number theoretic/combinatorial functions

```
divisible(x::fmpz, y::fmpz)
divisible(x::fmpz, y::Int)
```

Return `true` if x is divisible by y, otherwise return `false`. If $y = 0$ a `DivideError()` is raised.

```
issquare(x::fmpz)
```

Return `true` if x is a perfect square, otherwise return `false`.

```
isprime(x::fmpz)
```

Return `true` if x is prime, otherwise return `false`.

```
isprobabprime(x::fmpz)
```

Return `true` if x is very probably prime, otherwise return `false`.

```
remove(x::fmpz, y::fmpz)
```

Returns a pair (n, z) where n is an `Int` and z a `fmpz` such that $x = y^n z$ and z is not divisible by y. If $y = 0$ a `DivideError()` is thrown.

```
divisor_lenstra(n::fmpz, r::fmpz, m::fmpz)
```

If n has a factor which lies in the residue class $r \pmod m$ for $0 < r < m < n$, this function returns such a factor. Otherwise it returns 0. This is only efficient if m is at least the cube root of n. We require $\gcd(r, m) = 1$ and this condition is not checked.

```
fac(n::Int)
```

Return as a `fmpz` the factorial of n, i.e. $n(n-1)(n-2)\ldots 1$. If $n < 0$ we throw a `DomainError()`.

```
risingfac(x::fmpz, n::Int)
risingfac(x::Int, n::Int)
```

Return the rising factorial of x, i.e. $x(x+1)(x+2)\ldots(x+n-1)$. If $n < 0$ we throw a `DomainError()`.

```
primorial(n::Int)
```

Return as a `fmpz` the primorial of n, i.e. the product of all primes less than or equal to n. If $n < 0$ we throw a `DomainError()`.

```
fib(n::Int)
```

Return the n-th element of the Fibonacci sequence, starting with `fib(1) = 1`, `fib(2) = 1` and following the recursion `fib(n) = fib(n - 1) + fib(n - 2)` for $n \geq 3$. We return $\text{fib}(0) = 0$. If $n < 0$ we throw a `DomainError()`.

```
binom(n::Int, k::Int)
```

Return as a `fmpz` the binomial coefficient $\frac{n!}{(n-k)!k!}$. If $n, k < 0$ or $k > n$ we return 0.

```
moebiusmu(x::fmpz)
```

Returns the Möbius mu function of x as an `Int`. The value returned is either -1, 0 or 1. If $x < 0$ we throw a `DomainError()`.

```
jacobi(x::fmpz, y::fmpz)
```

Return the value of the Jacobi symbol $\left(\frac{x}{y}\right)$. If $y \leq x$ or $x < 0$ we throw a `DomainError()`.

```
sigma(x::fmpz, y::Int)
```

Return the value of the sigma function, i.e. $\sum_{0<d\,|x} d^y$. If $y < 0$ we throw a `DomainError()`.

```
eulerphi(x::fmpz)
```

Return the value of the Euler phi function, i.e. the number of $0 < d \leq x$ such that $\gcd(d, x) = 1$. If $x < 0$ we throw a `DomainError()`.

```
bell(x::Int)
```

Return the Bell number B_n.

```
numpart(x::fmpz)
numpart(x::Int)
```

Return the number of partitions of x, i.e. $p(x)$. Since $p(x)$ has $O(\sqrt{x})$ digits, this function can only be computed up to about $x = 10^{19}$. For large x, the result is computed faster if one sets `flint_set_num_threads(2)`.

Examples.

Here are some examples of number theoretic and combinatorial functions.

```
if isprime(fmpz(13))
   println("13 is prime")
end

n = fac(100)
s = sigma(fmpz(128), 10)
a = eulerphi(fmpz(12480))

p = numpart(1000)
```

8.3 Flint polynomials over $\mathbb{Z}$: `fmpz_poly`

The Flint integer polynomial type is `fmpz_poly`. Polynomials of this type are created using the `PolynomialRing` constructor.

```
R, x = PolynomialRing(ZZ, "x")
```

Polynomials in this ring have parent of type `FmpzPolyRing`.

We describe functions for `fmpz_poly` only where they differ from the functions available for generic polynomials.

8.3.1 Ad hoc binary operators

The following ad hoc binary operators are provided, as these are faster than first coercing to a common ring and then applying the operator.

```
+(x::fmpz_poly, y::Int)
+(x::fmpz_poly, y::fmpz)
+(x::Int, y::fmpz_poly)
+(x::fmpz, y::fmpz_poly)
```

Return the polynomial $x + y$.

```
-(x::fmpz_poly, y::Int)
-(x::fmpz_poly, y::fmpz)
-(x::Int, y::fmpz_poly)
-(x::fmpz, y::fmpz_poly)
```

Return the polynomial $x - y$.

Examples.

Here are some examples of ad hoc binary operators.

```
R, x = PolynomialRing(ZZ, "x")

f = x^3 + 2x + 1
g = x^2 - 7x + 4

h = f + 7
k = g - 7
l = fmpz(3) + f
```

8.3.2 Ad hoc comparisons

```
==(x::fmpz_poly, y::fmpz)
==(x::fmpz, y::fmpz_poly)
```

Return `true` if $x == y$, otherwise return `false`.

Examples.

Here are some examples of ad hoc comparisons.

```
R, x = PolynomialRing(ZZ, "x")

f = x^3 + 2x + 1

f != fmpz(6)
fmpz(7) == R(7)
```

8.3.3 Ad hoc exact division

```
divexact(x::fmpz_poly, y::Int)
```

Return the quotient of x by y, assuming the division is exact.

Examples.

Here is an example of ad hoc division.

```
R, x = PolynomialRing(ZZ, "x")

f = x^3 + 2x + 1

g = divexact(f*3, 3)
```

8.3.4 Content, primitive part and GCD

`content(f::fmpz_poly)`
Return the content of the polynomial f, i.e. the greatest common divisor of its coefficients.

`primpart(f::fmpz_poly)`

Return the primitive part of the polynomial f, i.e. the polynomial divided by its content.

`gcd(f::fmpz_poly, g::fmpz_poly)`

Return the greatest common divisor of the polynomials f and g.

Examples.

Here is an example of ad hoc division.

```
R, x = PolynomialRing(ZZ, "x")

f = x^2 + 2x + 1
g = x^3 + 3x + 1
h = x + 1

k = content(3*f)
l = primpart(3*f)
m = gcd(f*h, g*h)
```

8.3.5 Signature

`signature(f::fmpz_poly)`
Computes the signature of the polynomial f, i.e. a tuple (r, s) where r is the number of real roots and s is half the number of complex roots.

Examples.

Here is an example of signature.

```
R, x = PolynomialRing(ZZ, "x")

f = x^3 + 3x + 1

(r, s) = signature(f)
```

8.3.6 Special polynomials

The following functions compute univariate special polynomials over `fmpz`. It is expected that the final argument in each of these function is an element of the polynomial ring.

`chebyshev_t(n::Int, x::fmpz_poly)`

Return the Chebyshev polynomial of the first kind $T_n(x)$, defined by $T_n(x) = \cos(n \cos^{-1}(x))$.

`chebyshev_u(n::Int, x::fmpz_poly)`

Return the Chebyshev polynomial of the first kind $U_n(x)$, defined by $(n+1)U_n(x) = T'_{n+1}(x)$.

`cyclotomic(n::Int, x::fmpz_poly)`

Return the nth cyclotomic polynomial, defined as

$$\Phi_n(x) = \prod_{\omega}(x - \omega),$$

where ω runs over all the nth primitive roots of unity.

`swinnerton_dyer(n::Int, x::fmpz_poly)`

Return the Swinnerton-Dyer polynomial S_n, defined as the integer polynomial

$$S_n = \prod(x \pm \sqrt{2} \pm \sqrt{3} \pm \sqrt{5} \pm \ldots \pm \sqrt{p_n})$$

where p_n denotes the n-th prime number and all combinations of signs are taken. This polynomial has degree 2^n and is irreducible over the integers (it is the minimal polynomial of $\sqrt{2} + \ldots + \sqrt{p_n}$)

`cos_minpoly{S}(n::Int, x::fmpz_poly)`

Return the minimal polynomial of $2\cos(2\pi/n)$. For suitable choice of n, this gives the minimal polynomial of $2\cos(a\pi)$ or $2\sin(a\pi)$ for any rational a.

`theta_qexp{S}(r::Int, n::Int, x::fmpz_poly)`

Return the q-expansion to length n of the Jacobi theta function raised to the power r, i.e. $\vartheta(q)^r$ where $\vartheta(q) = 1 + \sum_{k=1}^{\infty} q^{k^2}$.

`eta_qexp{S}(r::Int, n::Int, x::fmpz_poly)`

Return the q-expansion to length n of the Dedekind eta function (without the leading factor $q^{1/24}$) raised to the power r, i.e. $(q^{-1/24}\eta(q))^r = \prod_{k=1}^{\infty}(1 - q^k)^r$.

In particular, $r = -1$ gives the generating function of the partition function $p(k)$, and $r = 24$ gives, after multiplication by q, the modular discriminant $\Delta(q)$ which generates the Ramanujan tau function $\tau(k)$.

Examples.

Here are some examples of computing special polynomials.

```
R, x = PolynomialRing(ZZ, "x")

f = chebyshev_t(20, x)
g = chebyshev_u(15, x)
h = cyclotomic(120, x)
j = swinnerton_dyer(5, x)
k = cos_minpoly(30, x)
l = theta_qexp(3, 30, x)
m = eta_qexp(24, 30, x)
o = cyclotomic(10, 1+x+x^2)
```

8.4 Flint polynomials over $\mathbb{Z}/n\mathbb{Z}$ (multiprecision n): fmpz_mod_poly

The Flint type for polynomials over $\mathbb{Z}/n\mathbb{Z}$ for multiprecision n is fmpz_mod_poly. Polynomials of this type are created using the PolynomialRing constructor. They are automatically created if the modulus n doesn't fit into a UInt.

```
R = ResidueRing(ZZ, 123456789012345678901)
S, x = PolynomialRing(R, "x")
```

Polynomials in this ring have parent of type FmpzModPolyRing.

We describe functions for fmpz_mod_poly only where they differ from the functions available for generic polynomials.

8.4.1 Constructors

In the constructors below S is assumed to be an FmpzModPolyRing as constructed above.

```
S(arr::Array{fmpz, 1})
```

> Construct the polynomial whose coefficients are those in the array, reduced modulo the modulus of the base ring of S.

```
S(a::fmpz)
```

> Construct the polynomial of degree 0 whose coefficient is the given integer reduced modulo the modulus of the base ring of S (or the zero polynomial if a is zero when reduced modulo the modulus).

```
S(p::fmpz_poly)
```

> Construct the polynomial whose coefficients are those of the given polynomial, but reduced modulo the modulus of the base ring of S.

Examples.

Here are some examples of constructors.

```
R = ResidueRing(ZZ, 123456789012345678901)
S, x = PolynomialRing(R, "x")

a = S()
b = S(x + 1)
c = S([R(1), R(0), R(2)])
d = S(c)
```

8.4.2 Ad hoc binary operators

```
*(x::fmpz_mod_poly, y::Residue{fmpz})
*(x::Residue{fmpz}, y::fmpz_mod_poly)
```

Return the polynomial xy.

```
+(x::fmpz_mod_poly, y::Residue{fmpz})
+(x::Residue{fmpz}, y::fmpz_mod_poly)
```

Return the polynomial $x + y$.

```
-(x::fmpz_mod_poly, y::Residue{fmpz})
-(x::Residue{fmpz}, y::fmpz_mod_poly)
```

Return the polynomial $x - y$.

Examples.

Here are some examples of ad hoc binary operators.

```
R = ResidueRing(ZZ, 123456789012345678949)
S, x = PolynomialRing(R, "x")

f = x^2 + 2x + 1
g = x^3 + 3x + 2

a = R(12) + g
b = f - R(12)
c = f*R(12)
```

8.4.3 Ad hoc comparison

```
==(x::fmpz_mod_poly, y::Residue{fmpz})
==(x::Residue{fmpz}, y::fmpz_mod_poly)
```

Return `true` if x and y are equal, otherwise return `false`.

Examples.

Here are some examples of ad hoc comparison.

```
R = ResidueRing(ZZ, 123456789012345678949)
S, x = PolynomialRing(R, "x")

f = x^2 + 2x + 1
g = x^3 + 3x + 2

S(7) == R(7)
R(7) != x + 1
```

8.4.4 Ad hoc exact division

`divexact(x::fmpz_mod_poly, y::Residue{fmpz})`
Divide the polynomial x by the residue y, assuming the division is exact.

Examples.

Here is an example of ad hoc exact division.

```
R = ResidueRing(ZZ, 123456789012345678949)
S, x = PolynomialRing(R, "x")

f = x^2 + 2x + 1

a = divexact(R(7)*f, R(7))
```

8.4.5 Modular arithmetic

`powmod(x::fmpz_mod_poly, e::fmpz, y::fmpz_mod_poly)`
Return $x^e \pmod{y}$.

Examples.

Here is an example of modular arithmetic.

```
R = ResidueRing(ZZ, 123456789012345678949)
S, x = PolynomialRing(R, "x")

f = x^2 + 2x + 1
g = x^3 + 3x + 1

a = powmod(f, fmpz(7), g)
```

8.4.6 Lifting

`lift(x::FmpzPolyRing, y::fmpz_mod_poly)`
Lift the polynomial y over $\mathbb{Z}/n\mathbb{Z}$ to a polynomial over $\mathbb{Z}$ in the given ring.

Examples.

Here is an example of lifting.

```
R = ResidueRing(ZZ, 123456789012345678949)
S, x = PolynomialRing(R, "x")

T, y = PolynomialRing(ZZ, "y")

f = x^2 + 2x + 1

a = lift(T, f)
```

8.4.7 Irreducibility testing

`isirreducible(x::fmpz_mod_poly)`
Return `true` if the polynomial x is irreducible, otherwise return `false`.

Examples.

Here is an example of lifting.

```
R = ResidueRing(ZZ, 123456789012345678949)
S, x = PolynomialRing(R, "x")

f = x^2 + 2x + 1

isirreducible(f) == false
```

8.4.8 Squarefree testing

`issquarefree(x::fmpz_mod_poly)`
Return `true` if the polynomial x is squarefree, otherwise return `false`.

Examples.

Here is an example of lifting.

```
R = ResidueRing(ZZ, 123456789012345678949)
S, x = PolynomialRing(R, "x")

f = x^2 + 2x + 1

issquarefree(f) == false
```

8.4.9 Factorisation

`factor(x::fmpz_mod_poly)`
Return an array containing the factorisation of the given polynomial. The entries in the array are tuples (p, e) consisting of irreducible polynomials p and their exponent e in the factorisation.

`factor_squarefree(x::fmpz_mod_poly)`

Return an array containing the squarefree factorisation of the given polynomial. The entries in the array are tuples (p, e) consisting of squarefree polynomials p and their exponent e in the factorisation.

`factor_distinct_deg(x::fmpz_mod_poly)`

Return an array containing the distinct degree factorisation of the given polynomial. The entries in the array are tuples (p, e) consisting of polynomials p which are the products of all the irreducible polynomials of the given degree e in the factorisation.

`factor_shape(x::fmpz_mod_poly)`

Return an array containing the shape of the factorisation of the given polynomial. The entries in the array are tuples (d, e) consisting of the degrees d of the irreducible polynomials and their exponents e in the factorisation.

Examples.

Here are some examples of factoring.

```
R = ResidueRing(ZZ, 123456789012345678949)
S, x = PolynomialRing(R, "x")

f = x^2 + 2x + 1
g = x^3 + 3x + 1

R = factor(f*g)
S = factor_squarefree(f*g)
T = factor_distinct_deg((x + 1)*g*(x^5+x+1))
U = factor_shape(f*g)
```

8.5 Flint polynomials over $\mathbb{Z}/n\mathbb{Z}$ (small n): `nmod_poly`

The Flint type for polynomials over $\mathbb{Z}/n\mathbb{Z}$ for small n is `nmod_poly`. Polynomials of this type are created using the `PolynomialRing` constructor. They are automatically created if the modulus n fits into a `UInt`.

```
R = ResidueRing(ZZ, 7)
S, x = PolynomialRing(R, "x")
```

Polynomials in this ring have parent of type `NmodPolyRing`.

We describe functions for `nmod_poly` only where they differ from the functions available for generic polynomials.

8.5.1 Constructors

In the constructors below S is assumed to be an `NmodPolyRing` as constructed above.

`S(arr::Array{UInt, 1})`

Construct the polynomial whose coefficients are those in the array.

```
S(arr::Array{fmpz, 1})
```

Construct the polynomial whose coefficients are those in the array, reduced modulo the modulus of the base ring of S.

```
S(a::UInt)
```

Construct the polynomial of degree 0 whose coefficient is the given integer (or the zero polynomial if a is zero).

```
S(a::fmpz)
```

Construct the polynomial of degree 0 whose coefficient is the given integer reduced modulo the modulus of the base ring of S (or the zero polynomial if a is zero when reduced modulo the modulus).

```
S(p::fmpz_poly)
```

Construct the polynomial whose coefficients are those of the given polynomial, but reduced modulo the modulus of the base ring of S.

Examples.

Here are some examples of constructors.

```
R = ResidueRing(ZZ, 7)
S, x = PolynomialRing(R, "x")

a = S()
b = S(x + 1)
c = S([R(1), R(0), R(2)])
d = S(c)
```

8.5.2 Ad hoc binary operators

```
*(x::nmod_poly, y::Residue{fmpz})
*(x::Residue{fmpz}, y::nmod_poly)
```

Return the polynomial xy.

```
+(x::nmod_poly, y::Residue{fmpz})
+(x::Residue{fmpz}, y::nmod_poly)
```

Return the polynomial $x + y$.

```
-(x::nmod_poly, y::Residue{fmpz})
-(x::Residue{fmpz}, y::nmod_poly)
```

Return the polynomial $x - y$.

Examples.

Here are some examples of ad hoc binary operators.

```
R = ResidueRing(ZZ, 7)
S, x = PolynomialRing(R, "x")

f = x^2 + 2x + 1
g = x^3 + 3x + 2

a = R(12) + g
b = f - R(12)
c = f*R(12)
```

8.5.3 Ad hoc comparison

`==(x::nmod_poly, y::Residue{fmpz})`
`==(x::Residue{fmpz}, y::nmod_poly)`

Return `true` if x and y are equal, otherwise return `false`.

Examples.

Here are some examples of ad hoc comparison.

```
R = ResidueRing(ZZ, 13)
S, x = PolynomialRing(R, "x")

f = x^2 + 2x + 1
g = x^3 + 3x + 2

S(7) == R(7)
R(7) != x + 1
```

8.5.4 Ad hoc exact division

`divexact(x::nmod_poly, y::Residue{fmpz})`

Divide the polynomial x by the residue y, assuming the division is exact.

Examples.

Here is an example of ad hoc exact division.

```
R = ResidueRing(ZZ, 23)
S, x = PolynomialRing(R, "x")

f = x^2 + 2x + 1

a = divexact(R(7)*f, R(7))
```

8.5.5 Interpolation

`divexact(R::NmodPolyRing, xval::Array{Residue{fmpz}, 1}, yval::Array{Residue{fmpz},`

Returns the unique polynomial in the given ring R of degree at most equal to the length of the two supplied arrays, which interpolates the values in the array `yval` at the values in the array `xval`.

Examples.

Here is an example of interpolation.

```
R = ResidueRing(ZZ, 23)
S, x = PolynomialRing(R, "x")

xval = [ R(0), R(1), R(2), R(3) ]

yval = [ R(0), R(1), R(4), R(9) ]

f = interpolate(S, xval, yval)
```

8.5.6 Inflation and deflation

`inflate(f::nmod_poly, n::Int)`

Given a polynomial $f(x)$, returns $f(x^n)$.

`deflate(f::nmod_poly, n::Int)`

Given a polynomial $f(x^n)$, returns $f(x)$.

Examples.

Here is an example of inflation and deflation.

```
R = ResidueRing(ZZ, 23)
S, x = PolynomialRing(R, "x")

f = x^6 + x^4 + 2 *x^2

g = inflate(f, 2)
h = deflate(g, 2)
```

8.5.7 Lifting

`lift(x::FmpzPolyRing, y::nmod_poly)`

Lift the polynomial y over $\mathbb{Z}/n\mathbb{Z}$ to a polynomial over $\mathbb{Z}$ in the given ring.

Examples.

Here is an example of lifting.

```
R = ResidueRing(ZZ, 23)
S, x = PolynomialRing(R, "x")

T, y = PolynomialRing(ZZ, "y")

f = x^2 + 2x + 1

a = lift(T, f)
```

8.5.8 Irreducibility testing

`isirreducible(x::nmod_poly)`
Return `true` if the polynomial x is irreducible, otherwise return `false`.

Examples.

Here is an example of lifting.

```
R = ResidueRing(ZZ, 23)
S, x = PolynomialRing(R, "x")

f = x^2 + 2x + 1

isirreducible(f) == false
```

8.5.9 Squarefree testing

`issquarefree(x::nmod_poly)`
Return `true` if the polynomial x is squarefree, otherwise return `false`.

Examples.

Here is an example of lifting.

```
R = ResidueRing(ZZ, 23)
S, x = PolynomialRing(R, "x")

f = x^2 + 2x + 1

issquarefree(f) == false
```

8.5.10 Factorisation

`factor(x::nmod_poly)`
Return an array containing the factorisation of the given polynomial. The entries in the array are tuples (p, e) consisting of irreducible polynomials p and their exponent e in the factorisation.

`factor_squarefree(x::nmod_poly)`

Return an array containing the squarefree factorisation of the given polynomial. The entries in the array are tuples (p, e) consisting of squarefree polynomials p and their exponent e in the factorisation.

```
factor_distinct_deg(x::nmod_poly)
```

Return an array containing the distinct degree factorisation of the given polynomial. The entries in the array are tuples (p, e) consisting of polynomials p which are the products of all the irreducible polynomials of the given degree e in the factorisation.

```
factor_shape(x::nmod_poly)
```

Return an array containing the shape of the factorisation of the given polynomial. The entries in the array are tuples (d, e) consisting of the degrees d of the irreducible polynomials and their exponents e in the factorisation.

Examples.

Here are some examples of factoring.

```
R = ResidueRing(ZZ, 23)
S, x = PolynomialRing(R, "x")

f = x^2 + 2x + 1
g = x^3 + 3x + 1

R = factor(f*g)
S = factor_squarefree(f*g)
T = factor_distinct_deg((x + 1)*g*(x^5+x^3+x+1))
U = factor_shape(f*g)
```

8.6 Flint polynomials over $\mathbb{Q}$: `fmpq_poly`

The Flint type for polynomials over $\mathbb{Q}$ is `fmpq_poly`. Polynomials of this type are created using the `PolynomialRing` constructor with `FlintRationalField()` as an argument (the same as `QQ`, by default).

```
S, x = PolynomialRing(QQ, "x")
```

Polynomials in this ring have parent of type `FmpqPolyRing`.

We describe functions for `fmpq_poly` only where they differ from the functions available for generic polynomials.

8.6.1 Constructors

In the following constructors we assume that S is an `FmpqPolyRing` parent object, as created by the above construction for example.

```
S(a::Int)
S(a::fmpz)
```

Construct the polynomial of degree 0 with a as its only coefficient (or the zero polynomial if $a = 0$).

```
S(a::fmpz_poly)
```

Coerce the given polynomial into the ring S, i.e. coerce its coefficients into $\mathbb{Q}$ and change the variable to that of S.

Examples.

Here are some examples of constructors.

```
S, x = PolynomialRing(QQ, "x")

a = S(12)
b = S(fmpz(12))

R, y = PolynomialRing(ZZ, "y")

c = S(3y^3 + 2y + 1)
```

8.6.2 Basic manipulation

Internally, Flint stores `fmpq_poly`'s as an array of integers and a single common denominator.

```
den(a::fmpq_poly)
```

Return the common denominator of the coefficients of the polynomial a.

Examples.

Here is an example of basic manipulation.

```
S, x = PolynomialRing(QQ, "x")

a = den(-fmpz(12)//7*x + 1)
```

8.6.3 Ad hoc binary operators

The following ad hoc binary operators are provided over and above the basic ones for performance reasons.

```
*(a::fmpq_poly, b::fmpq)
*(a::fmpq, b::fmpq_poly)
```

Return ab.

```
+(a::fmpq_poly, b::Int)
+(a::Int, b::fmpq_poly)
+(a::fmpq_poly, b::fmpz)
+(a::fmpz, b::fmpq_poly)
+(a::fmpq_poly, b::fmpq)
+(a::fmpq, b::fmpq_poly)
```

Return $a + b$.

```
-(a::fmpq_poly, b::Int)
-(a::Int, b::fmpq_poly)
-(a::fmpq_poly, b::fmpz)
-(a::fmpz, b::fmpq_poly)
-(a::fmpq_poly, b::fmpq)
-(a::fmpq, b::fmpq_poly)
```

Return $a - b$.

Examples.

Here are some examples of ad hoc binary operators.

```
S, x = PolynomialRing(QQ, "x")

a = 7y^2 + 3y + 2

b = a + 7
c = fmpz(7) - a
d = fmpq(2, 3)*a
```

8.6.4 Ad hoc comparisons

The following ad hoc binary comparisons are provided over and above the basic ones for performance reasons.

```
==(a::fmpq_poly, b::fmpq)
==(a::fmpq, b::fmpq_poly)
```

Return `true` if a is equal to b, otherwise return `false`.

Examples.

Here are some examples of ad hoc comparisons.

```
S, x = PolynomialRing(QQ, "x")

a = 7y^2 + 3y + 2

a != fmpq(2, 3)
fmpq(2, 3) == S(fmpq(2, 3))
```

8.6.5 Ad hoc exact division

```
divexact(a::fmpq_poly, b::Int)
divexact(a::fmpq_poly, b::fmpz)
```

Return the quotient of a by b. A `DivideError()` is raised if $b = 0$.

Examples.

Here are some examples of ad hoc exact division.

```
S, x = PolynomialRing(QQ, "x")

a = 7y^2 + 3y + 2

b = divexact(a, 7)
c = divexact(a, fmpz(11))
```

8.6.6 Signature

`signature(f::fmpq_poly)`

Computes the signature of the polynomial f, i.e. a tuple (r, s) where r is the number of real roots and s is half the number of complex roots.

Examples.

Here is an example of signature.

```
R, x = PolynomialRing(QQ, "x")

f = (x^3 + 3x + QQ(2)//QQ(3))

(r, s) = signature(f)
```

8.7 Flint polynomials over $\mathbb{F}_{p^k}$ (multiprecision p): `fq_poly`

The Flint type for polynomials over $\mathbb{F}_{p^k}$ for multiprecision p is `fq_poly`. Polynomials of this type are created using the `PolynomialRing` constructor with an `FqFiniteField` parent object as parameter. For example

```
R, x = FiniteField(fmpz(23), 5, "x")
S, y = PolynomialRing(R, "y")
```

Polynomials in this ring have parent of type `FqPolyRing`.

We describe functions for `fq_poly` only where they differ from the functions available for generic polynomials.

8.7.1 Constructors

In the following constructors we assume that S is an `FqPolyRing` parent object, as created by the above construction for example.

`S(a::fmpz)`

Construct the polynomial of degree 0 with a as its only coefficient (or the zero polynomial if $a = 0$).

```
S(a::Array{fmpz, 1})
S{T <: Integer}(a::Array{T, 1})
```

Construct the polynomial whose coefficients are the integers in the given array (thought of as elements of the finite field).

```
S(a::fmpz_poly)
```

Coerce the given polynomial into the ring S, i.e. coerce its coefficients into the finite field and change the variable to that of S.

Examples.

Here are some examples of constructors.

```
R, x = FiniteField(fmpz(23), 5, "x")
S, y = PolynomialRing(R, "y")

a = S(fmpz(12))
b = S([1, 2, 3])
c = S(3y^3 + 2y + 1)
```

8.7.2 Ad hoc binary operators

The following ad hoc binary operators are provided over and above the basic ones for performance reasons.

```
*(a::fq_poly, b::fq)
*(a::fq, b::fq_poly)
*(a::fq_poly, b::Integer)
*(a::Integer, b::fq_poly)
```

Return ab.

```
+(a::fq_poly, b::Int)
+(a::Int, b::fq_poly)
+(a::fq_poly, b::Integer)
+(a::Integer, b::fq_poly)
+(a::fq_poly, b::fmpz)
+(a::fmpz, b::fq_poly)
+(a::fq_poly, b::fq)
+(a::fq, b::fq_poly)
```

Return $a + b$.

```
-(a::fq_poly, b::Int)
-(a::Int, b::fq_poly)
-(a::fq_poly, b::Integer)
-(a::Integer, b::fq_poly)
-(a::fq_poly, b::fmpz)
-(a::fmpz, b::fq_poly)
-(a::fq_poly, b::fq)
-(a::fq, b::fq_poly)
```

Return $a - b$.

Examples.

Here are some examples of ad hoc binary operators.

```
R, x = FiniteField(fmpz(23), 5, "x")
S, y = PolynomialRing(R, "y")

a = 7y^2 + 3y + 2

b = a + 7
c = fmpz(7) - a
d = (x + 1)*a
```

8.7.3 Ad hoc comparisons

The following ad hoc binary comparisons are provided over and above the basic ones for performance reasons.

```
==(a::fq_poly, b::fq)
==(a::fq, b::fq_poly)
==(a::fq_poly, b::fmpz)
==(a::fmpz, b::fq_poly)
```

Return `true` if a is equal to b, otherwise return `false`.

Examples.

Here are some examples of ad hoc comparisons.

```
R, x = FiniteField(fmpz(23), 5, "x")
S, y = PolynomialRing(R, "y")

a = 7y^2 + 3y + 2

a != x + 1
x + 1 == S(x + 1)
a != fmpz(11)
```

8.7.4 Inflation and deflation

`inflate(f::fq_poly, n::Int)`

Given a polynomial $f(x)$, returns $f(x^n)$.

`deflate(f::fq_poly, n::Int)`

Given a polynomial $f(x^n)$, returns $f(x)$.

Examples.

Here is an example of inflation and deflation.

```
R, x = FiniteField(fmpz(23), 5, "x")
S, y = PolynomialRing(R, "y")

a = 7y^2 + 3y + 2

g = inflate(f, 2)
h = deflate(g, 2)
```

8.7.5 Factorisation

`factor(x::fq_poly)`

Return an array containing the factorisation of the given polynomial. The entries in the array are tuples (p, e) consisting of irreducible polynomials p and their exponent e in the factorisation.

`factor_distinct_deg(x::fq_poly)`

Return an array containing the distinct degree factorisation of the given polynomial. The entries in the array are tuples (p, e) consisting of polynomials p which are the products of all the irreducible polynomials of the given degree e in the factorisation.

Examples.

Here are some examples of factoring.

```
R, x = FiniteField(fmpz(23), 5, "x")
S, y = PolynomialRing(R, "y")

f = 7y^2 + 3y + 2
g = 11y^3 - 2y^2 + 5

A = factor(f*g)
B = factor_distinct_deg((y + 1)*g*(y^5+y^3+y+1))
```

8.8 Flint polynomials over $\mathbb{F}_{p^k}$ (small p): `fq_nmod_poly`

The Flint type for polynomials over $\mathbb{F}_{p^k}$ for p that fits in an `Int` is `fq_poly`. Polynomials of this type are created using the `PolynomialRing` constructor with an `FqNmodFiniteField` parent object as parameter. For example

```
R, x = FiniteField(23, 5, "x")
S, y = PolynomialRing(R, "y")
```

Polynomials in this ring have parent of type `FqNmodPolyRing`.

We describe functions for `fq_nmod_poly` only where they differ from the functions available for generic polynomials.

8.8.1 Constructors

In the following constructors we assume that S is an FqNmodPolyRing parent object, as created by the above construction for example.

```
S(a::fmpz)
```

Construct the polynomial of degree 0 with a as its only coefficient (or the zero polynomial if $a = 0$).

```
S(a::Array{fmpz, 1})
S{T <: Integer}(a::Array{T, 1})
```

Construct the polynomial whose coefficients are the integers in the given array (thought of as elements of the finite field).

```
S(a::fmpz_poly)
```

Coerce the given polynomial into the ring S, i.e. coerce its coefficients into the finite field and change the variable to that of S.

Examples.

Here are some examples of constructors.

```
R, x = FiniteField(23, 5, "x")
S, y = PolynomialRing(R, "y")

a = S(fmpz(12))
b = S([1, 2, 3])
c = S(3y^3 + 2y + 1)
```

8.8.2 Ad hoc binary operators

The following ad hoc binary operators are provided over and above the basic ones for performance reasons.

```
*(a::fq_nmod_poly, b::fq_nmod)
*(a::fq_nmod, b::fq_poly_nmod)
*(a::fq_nmod_poly, b::Integer)
*(a::Integer, b::fq_nmod_poly)
```

Return ab.

```
+(a::fq_nmod_poly, b::Int)
+(a::Int, b::fq_nmod_poly)
+(a::fq_nmod_poly, b::Integer)
+(a::Integer, b::fq_nmod_poly)
+(a::fq_nmod_poly, b::fmpz)
+(a::fmpz, b::fq_nmod_poly)
+(a::fq_nmod_poly, b::fq_nmod)
+(a::fq_nmod, b::fq_nmod_poly)
```

Return $a + b$.

```
-(a::fq_nmod_poly, b::Int)
-(a::Int, b::fq_nmod_poly)
-(a::fq_nmod_poly, b::Integer)
-(a::Integer, b::fq_nmod_poly)
-(a::fq_nmod_poly, b::fmpz)
-(a::fmpz, b::fq_nmod_poly)
-(a::fq_nmod_poly, b::fq_nmod)
-(a::fq_nmod, b::fq_nmod_poly)
```

Return $a - b$.

Examples.

Here are some examples of ad hoc binary operators.

```
R, x = FiniteField(23, 5, "x")
S, y = PolynomialRing(R, "y")

a = 7y^2 + 3y + 2

b = a + 7
c = fmpz(7) - a
d = (x + 1)*a
```

8.8.3 Ad hoc comparisons

The following ad hoc binary comparisons are provided over and above the basic ones for performance reasons.

```
==(a::fq_nmod_poly, b::fq_nmod)
==(a::fq_nmod, b::fq_nmod_poly)
==(a::fq_nmod_poly, b::fmpz)
==(a::fmpz, b::fq_nmod_poly)
```

Return `true` if a is equal to b, otherwise return `false`.

Examples.

Here are some examples of ad hoc comparisons.

```
R, x = FiniteField(23, 5, "x")
S, y = PolynomialRing(R, "y")

a = 7y^2 + 3y + 2

a != x + 1
x + 1 == S(x + 1)
a != fmpz(11)
```

8.8.4 Inflation and deflation

`inflate(f::fq_nmod_poly, n::Int)`
Given a polynomial $f(x)$, returns $f(x^n)$.

`deflate(f::fq_nmod_poly, n::Int)`

Given a polynomial $f(x^n)$, returns $f(x)$.

Examples.

Here is an example of inflation and deflation.

```
R, x = FiniteField(23, 5, "x")
S, y = PolynomialRing(R, "y")

a = 7y^2 + 3y + 2

g = inflate(f, 2)
h = deflate(g, 2)
```

8.8.5 Factorisation

`factor(x::fq_nmod_poly)`
Return an array containing the factorisation of the given polynomial. The entries in the array are tuples (p, e) consisting of irreducible polynomials p and their exponent e in the factorisation.

`factor_distinct_deg(x::fq_nmod_poly)`

Return an array containing the distinct degree factorisation of the given polynomial. The entries in the array are tuples (p, e) consisting of polynomials p which are the products of all the irreducible polynomials of the given degree e in the factorisation.

Examples.

Here are some examples of factoring.

```
R, x = FiniteField(23, 5, "x")
S, y = PolynomialRing(R, "y")

f = 7y^2 + 3y + 2
g = 11y^3 - 2y^2 + 5

A = factor(f*g)
B = factor_distinct_deg((y + 1)*g*(y^5+y^3+y+1))
```

8.9 Flint power series over $\mathbb{Z}$: fmpz_series

Flint doesn't have a module for power series over $\mathbb{Z}$, but the fmpz_poly module contains code for power series operations. In Nemo, for the purposes of simplicity, we deal with Flint as though it has a module called fmpz_series.

The power series type in Nemo for polynomials treated as power series is fmpz_series.

Power series of this type are created using the PowerSeriesRing constructor. It takes a maximum (relative) precision and a variable name as a string.

```
R, x = PowerSeriesRing(ZZ, 30, "x")
```

Power series in this ring have parent of type FmpzSeriesRing.

Currently the fmpz_series module provides nothing additional on top of the functionality that is described for generic power series.

8.10 Flint power series over $\mathbb{Z}/n\mathbb{Z}$ (multiprecision n): fmpz_mod_series

Flint doesn't have a module for power series over $\mathbb{Z}/n\mathbb{Z}$, but the fmpz_mod_poly module contains code for power series operations. In Nemo, for the purposes of simplicity, we deal with Flint as though it has a module called fmpz_mod_series.

The power series type in Nemo for polynomials treated as power series is fmpz_mod_series.

Power series of this type are created using the PowerSeriesRing constructor. It takes a maximum (relative) precision and a variable name as a string.

```
R = ResidueRing(ZZ, 123456789012345678949)
S, x = PowerSeriesRing(R, 30, "x")
```

Power series in this ring have parent of type FmpzModSeriesRing.

We list only the functionality that the fmpz_mod_series module provides on top of the functionality that is described for generic power series.

8.10.1 Constructors

In the following we assume that S is a power series ring parent object as created by the construction above, for example.

```
S(a::Array{Residue{fmpz}, 1}, n::Int, p::Int)
```

> Create the power series in the ring S whose coefficients are the entries in the array a, reduced modulo the modulus of the residue ring. The length of the array is specified by n, though n can be less than this (remaining entries in the array are ignored) and p gives the precision of the power series created.

Examples.

Here are some examples of power series constructors.

```
R = ResidueRing(ZZ, 123456789012345678949)
S, x = PowerSeriesRing(R, 30, "x")

f = S([R(0), R(3), R(1)], 3, 5)
```

8.11 Flint power series over $\mathbb{Q}$: fmpq_series

Flint doesn't have a module for power series over $\mathbb{Q}$, but the fmpq_poly module contains code for power series operations. In Nemo, for the purposes of simplicity, we deal with Flint as though it has a module called fmpq_series.

The power series type in Nemo for polynomials treated as power series is fmpq_series.

Power series of this type are created using the PowerSeriesRing constructor. It takes a maximum (relative) precision and a variable name as a string.

```
R, x = PowerSeriesRing(QQ, 30, "x")
```

Power series in this ring have parent of type FmpqSeriesRing.

We describe only the functions which are different to those described for generic power series.

8.11.1 Ad hoc binary operators

```
*(x::fmpq_series, y::fmpq)
*(x::fmpq, y::fmpq_series)
```

Return xy.

Examples.

Here are some examples of ad hoc binary operators.

```
R, x = PowerSeriesRing(QQ, 30, "x")

a = 1 + x + 3x^2 + O(x^5)

b = a*fmpq(2, 3)

c = fmpq(2, 3)*a
```

8.11.2 Special functions

```
exp(x::fmpq_series)
```

Compute the exponential of the given power series. The constant term must be 0.

```
log(x::fmpq_series)
```

Compute the logarithm of the given power series. The constant term must be 1.

```
sin(x::fmpq_series)
cos(x::fmpq_series)
tan(x::fmpq_series)
asin(x::fmpq_series)
atan(x::fmpq_series)
```

Compute the respective trigonometric series by substituting the given power series. The constant term must be 0.

```
sinh(x::fmpq_series)
cosh(x::fmpq_series)
tanh(x::fmpq_series)
asinh(x::fmpq_series)
atanh(x::fmpq_series)
```

Compute the respective hyperbolic trigonometric series by substituting the given power series. The constant term must be 0.

```
sqrt(x::fmpq_series)
```

Compute the square root of the given power series. The constant term must be 1.

Examples.

Here are some examples of ad hoc binary operators.

```
R, x = PowerSeriesRing(QQ, 30, "x")

a = 1 + x + 3x^2 + O(x^5)
b = x + 2x^2 + 5x^3 + O(x^5)

c = exp(b)
d = log(a)
f = sqrt(a)
g = sin(b)
h = atanh(b)
```

8.12 Flint power series over $\mathbb{F}_{p^k}$ (multiprecision p): fq_series

Flint doesn't have a module for power series over $\mathbb{F}_{p^k}$, but the fq_poly module contains code for power series operations. In Nemo, for the purposes of simplicity, we deal with Flint as though it has a module called fq_series.

The power series type in Nemo for polynomials treated as power series is fq_series.

Power series of this type are created using the PowerSeriesRing constructor when it is passed an FqFiniteField parent object as argument. The function also takes a maximum (relative) precision and a variable name as a string.

```
R, t = FiniteField(fmpz(23), 5, "t")
S, x = PowerSeriesRing(R, 30, "x")
```

Power series in this ring have parent of type FqSeriesRing.

We describe only the functions which are different to those described for generic power series.

8.12.1 Constructors

In the constructor below we assume S is an FqSeriesRing parent object, as constructed above for example. We assume the maximum precision of the power series for this object to be k.

```
S(a::fmpz)
```

Construct the power series $a + O(x^k)$.

Examples.

Here is an example of a constructor.

```
R, t = FiniteField(fmpz(23), 5, "t")
S, x = PowerSeriesRing(R, 30, "x")

a = S(fmpz(12))
```

8.12.2 Ad hoc binary operators

```
*(x::fq_series, y::fq)
*(x::fq, y::fq_series)
```

Return xy.

Examples.

Here are some examples of ad hoc binary operators.

```
R, t = FiniteField(fmpz(23), 5, "t")
S, x = PowerSeriesRing(R, 30, "x")

a = 1 + x + 3x^2 + O(x^5)

b = a*(t^2 + 1)

c = 2t*a
```

8.12.3 Ad hoc exact division

```
divexact(x::fq_series, y::fq)
```

Return x/y. If $y = 0$ this raises a `DivideError()`.

Examples.

Here is an example of ad hoc exact division.

```
R, t = FiniteField(fmpz(23), 5, "t")
S, x = PowerSeriesRing(R, 30, "x")

a = 1 + x + 3x^2 + O(x^5)

b = divexact(a, t^2 + 1)
```

8.13 Flint power series over $\mathbb{F}_{p^k}$ (small p): fq_nmod_series

Flint doesn't have a module for power series over $\mathbb{F}_{p^k}$ for small p, but the fq_nmod_poly module contains code for power series operations. In Nemo, for the purposes of simplicity, we deal with Flint as though it has a module called fq_nmod_series.

The power series type in Nemo for polynomials treated as power series is fq_nmod_series.

Power series of this type are created using the PowerSeriesRing constructor when it is passed an FqNmodFiniteField parent object as argument. The function also takes a maximum (relative) precision and a variable name as a string.

```
R, t = FiniteField(23, 5, "t")
S, x = PowerSeriesRing(R, 30, "x")
```

Power series in this ring have parent of type FqNmodSeriesRing.

We describe only the functions which are different to those described for generic power series.

8.13.1 Constructors

In the constructor below we assume S is an FqNmodSeriesRing parent object, as constructed above for example. We assume the maximum precision of the power series for this object to be k.

```
S(a::fmpz)
```

Construct the power series $a + O(x^k)$.

Examples.

Here is an example of a constructor.

```
R, t = FiniteField(23, 5, "t")
S, x = PowerSeriesRing(R, 30, "x")

a = S(fmpz(12))
```

8.13.2 Ad hoc binary operators

```
*(x::fq_nmod_series, y::fq_nmod)
*(x::fq_nmod, y::fq_nmod_series)
```

Return xy.

Examples.

Here are some examples of ad hoc binary operators.

```
R, t = FiniteField(23, 5, "t")
S, x = PowerSeriesRing(R, 30, "x")

a = 1 + x + 3x^2 + O(x^5)

b = a*(t^2 + 1)

c = 2t*a
```

8.13.3 Ad hoc exact division

```
divexact(x::fq_nmod_series, y::fq_nmod)
```

Return x/y. If $y = 0$ this raises a `DivideError()`.

Examples.

Here is an example of ad hoc exact division.

```
R, t = FiniteField(23, 5, "t")
S, x = PowerSeriesRing(R, 30, "x")

a = 1 + x + 3x^2 + O(x^5)

b = divexact(a, t^2 + 1)
```

8.14 Flint matrices over $\mathbb{Z}/n\mathbb{Z}$ (small n): `nmod_mat`

The Flint type for matrices over $\mathbb{Z}/n\mathbb{Z}$ for small n is `nmod_mat`. Matrices of this type are created using the `MatrixSpace` constructor in Nemo. They are automatically created if the modulus n fits into a `UInt`.

```
R = ResidueRing(ZZ, 7)
S = MatrixSpace(R, 3, 3)
```

Matrices in this matrix space have parent of type `NmodMatSpace`.

We describe functions for `nmod_mat` only where they differ from the functions available for generic matrices.

8.14.1 Constructors

In the constructors below S is assumed to be an `NmodMatSpace` as constructed above. We assume that r is the number of rows and c the number of columns for matrices in this space, e.g. $r = 3$ and $c = 3$ in the example above.

```
S(a::Array{BigInt, 2})
```

Construct the $r \times c$ matrix with entries given by the Julia array a, viewed as elements of the base ring.

```
S(arr::Array{Int, 1})
S(arr::Array{BigInt, 1})
S(arr::Array{fmpz, 1})
S(arr::Array{Residue{fmpz}, 1})
```

Given a one dimensional Julia array of length rc, construct the matrix with r rows and c columns with the array entries as matrix entries. The entries are specified one row after the other.

Examples.

Here are some examples of constructors.

```
R = ResidueRing(ZZ, 7)
S = MatrixSpace(R, 3, 3)

a = S([BigInt(4) 5 6; 7 3 2; 1 4 5])
b = S([4, 5, 6, 7, 3, 2, 1, 4, 5])
c = S([R(4), R(5), R(6), R(7), R(3), R(2), R(1), R(4), R(5)])
```

8.14.2 Ad hoc binary operators

`*(a::nmod_mat, b::UInt)`

Return ab, i.e. the matrix whose entries are those of a multiplied by b.

`*(a::UInt, b::nmod_mat)`

Return ab, i.e. the matrix whose entries are a multiplied by the entries of b.

Examples.

Here is an example of ad hoc binary operators.

```
R = ResidueRing(ZZ, 7)
S = MatrixSpace(R, 3, 3)

a = S([4 5 6; 7 3 2; 1 4 5])

b = UInt(5)*a
```

8.14.3 Row echelon form

`rref(a::nmod_mat)`

Return the row echelon form of a.

`rref!(a::nmod_mat)`

Compute the row echelon form of a in-place. It is the users responsibility to ensure this matrix is not aliased elsewhere.

Examples.

Here is an example of row echelon form.

```
R = ResidueRing(ZZ, 7)
S = MatrixSpace(R, 3, 3)

a = S([4 5 6; 7 3 2; 1 4 5])

b = rref(a)
```

8.14.4 Determinant

`determinant(a::nmod_mat)`
Return the determinant of a as an `mpzResiduef`.

Examples.

Here is an example of determinant.

```
R = ResidueRing(ZZ, 7)
S = MatrixSpace(R, 3, 3)

a = S([4 5 6; 7 3 2; 1 4 5])

b = determinant(a)
```

8.14.5 Rank

`rank(a::nmod_mat)`
Return the rank of a.

Examples.

Here is an example of computing the rank.

```
R = ResidueRing(ZZ, 7)
S = MatrixSpace(R, 3, 3)

a = S([4 5 6; 7 3 2; 1 4 5])

b = rank(a)
```

8.14.6 Inversion

`inv(a::nmod_mat)`
Return the matrix inverse of a. Note the matrix must be must be square and invertible, otherwise an exception is thrown.

Examples.

Here is an example of inversion.

```
R = ResidueRing(ZZ, 7)
S = MatrixSpace(R, 3, 3)

a = S([BigInt(4) 5 6; 7 3 2; 1 4 5])

b = inv(a)
```

8.14.7 Linear solving

`solve(a::nmod_mat, b::nmod_mat)`

Return the column vector x such that $ax = b$ where x and b are column vectors with the same number of rows as the a. Note that a must be a square matrix. If these conditions are not met, an exception is raised.

Examples.

Here is an example of linear solving.

```
R = ResidueRing(ZZ, 7)
S = MatrixSpace(R, 3, 3)
T = MatrixSpace(R, 3, 1)

a = S([4 5 6; 7 3 2; 1 4 5])
b = T([5, 3, 2])

x = solve(a, b)
```

8.14.8 LU decomposition

`lufact(a::nmod_mat)`

Compute a tuple of matrices (l, u, p) such that $pa = lu$ where l is a lower triangular matrix, u is upper triangular and p is a permutation matrix.

Examples.

Here is an example of LU decomposition.

```
R = ResidueRing(ZZ, 7)
S = MatrixSpace(R, 3, 3)
T = MatrixSpace(R, 3, 1)

a = S([4 5 6; 7 3 2; 1 4 5])

l, u, p = lufact(a)
```

8.14.9 Matrix concatenation

`hcat(a::nmod_mat, b::nmod_mat)`

Return the matrix that is the horizontal concatenation of a and b. Both a and b must have the same number of rows, else an exception is raised.

`vcat(a::nmod_mat, b::nmod_mat)`

Return the matrix that is the vertical concatenation of a and b. Both a and b must have the same number of columns, else an exception is raised.

Examples.

Here are some examples of concatenation.

```
R = ResidueRing(ZZ, 7)
S = MatrixSpace(R, 3, 3)
T = MatrixSpace(R, 3, 1)

a = S([4 5 6; 7 3 2; 1 4 5])
b = S([9 5 3; 2 7 1; 0 6 9])

c = hcat(a, b)
d = vcat(a, b)
```

8.14.10 Conversions

`Array(a::nmod_mat)`

Return a two dimensional Julia array with entries that are of type `mpzResiduef` containing the entries of the matrix a.

Examples.

Here is an example of conversion.

```
R = ResidueRing(ZZ, 7)
S = MatrixSpace(R, 3, 3)
T = MatrixSpace(R, 3, 1)

a = S([4 5 6; 7 3 2; 1 4 5])

A = Array(a)
```

8.14.11 Lifting

`lift(a::nmod_mat)`

Return a lift of the matrix a to a matrix over $\mathbb{Z}$, i.e. where the entries of the returned matrix are those of a lifted to $\mathbb{Z}$.

Examples.

Here is an example of lifting.

```
R = ResidueRing(ZZ, 7)
S = MatrixSpace(R, 3, 3)
T = MatrixSpace(R, 3, 1)

a = S([4 5 6; 7 3 2; 1 4 5])

b = lift(a)
```

8.15 Flint matrices over $\mathbb{Z}$: fmpz_mat

The Flint type for matrices over $\mathbb{Z}$ is fmpz_mat. Matrices of this type are created using the MatrixSpace constructor in Nemo.

```
S = MatrixSpace(ZZ, 3, 3)
```

Matrices in this matrix space have parent of type FmpzMatSpace.

We describe functions for fmpz_mat only where they differ from the functions available for generic matrices.

8.15.1 Row echelon form

rref(a::fmpz_mat)
: Return a tuple (R, d) where R/d is the row echelon form of a over $\mathbb{Q}$. Here R is a matrix over $\mathbb{Z}$ and d is an fmpz denominator.

Examples.

Here is an example of row echelon form.

```
S = MatrixSpace(ZZ, 3, 3)

A = S([fmpz(2) 3 5; 1 4 7; 4 1 1])

rref(A)
```

8.15.2 Determinant

determinant(a::fmpz_mat)
: Return the determinant of a.

Examples.

Here is an example of determinant.

```
S = MatrixSpace(ZZ, 3, 3)

a = S([4 5 6; 7 3 2; 1 4 5])

b = determinant(a)
```

8.15.3 Rank

rank(a::fmpz_mat)
: Return the rank of a.

Examples.

Here is an example of computing the rank.

```
S = MatrixSpace(ZZ, 3, 3)

a = S([4 5 6; 7 3 2; 1 4 5])

b = rank(a)
```

8.15.4 Inversion

`inv(a::fmpz_mat)`

Return the matrix inverse of a. Note the matrix must be must be square and invertible, otherwise an exception is thrown.

Examples.

Here is an example of inversion.

```
S = MatrixSpace(ZZ, 3, 3)

A = S([fmpz(2) 3 5; 1 4 7; 9 2 2])

B = inv(A)
```

8.15.5 Pseudo inversion

`pseudo_inv(a::fmpz_mat)`

Return a tuple (R, d) where R/d is the inverse of a over $\mathbb{Q}$. Here R is a matrix over $\mathbb{Z}$ and d is an `fmpz` denominator. Note that the matrix must be square and and non-singular, otherwise an exception is thrown.

Examples.

Here is an example of inversion.

```
S = MatrixSpace(ZZ, 3, 3)

A = S([1 0 1; 2 3 1; 5 6 7])

B, d = pseudo_inv(A)
```

8.15.6 Linear solving

`solve(a::fmpz_mat, b::fmpz_mat)`

Return a tuple (x, d) consisting of the column vector x such that $ax/d = b$ where x and b are column vectors with the same number of rows as the a and d is a denominator. Note that a must be a square matrix. If these conditions are not met, an exception is raised.

`solve_dixon(a::fmpz_mat, b::fmpz_mat)`

Return a tuple (x, m) consisting of the column vector x such that $ax = b \pmod{m}$ where x and b are column vectors with the same number of rows as the a. Note that a must be a square matrix. If these conditions are not met, an exception is raised.

Examples.

Here is an example of linear solving.

```
S = MatrixSpace(ZZ, 3, 3)
T = MatrixSpace(ZZ, 3, 1)

A = S([fmpz(2) 3 5; 1 4 7; 9 2 2])
B = T([fmpz(4), 5, 7])

X, d = solve(A, B)
X, m = solve_dixon(A, B)
```

8.15.7 Matrix concatenation

`hcat(a::fmpz_mat, b::fmpz_mat)`

Return the matrix that is the horizontal concatenation of a and b. Both a and b must have the same number of rows, else an exception is raised.

`vcat(a::fmpz_mat, b::fmpz_mat)`

Return the matrix that is the vertical concatenation of a and b. Both a and b must have the same number of columns, else an exception is raised.

Examples.

Here are some examples of concatenation.

```
S = MatrixSpace(ZZ, 3, 3)
T = MatrixSpace(ZZ, 3, 6)
U = MatrixSpace(ZZ, 6, 3)

A = S([fmpz(2) 3 5; 1 4 7; 9 6 3])
B = S([fmpz(1) 4 7; 9 6 7; 4 3 3])

C = hcat(A, B)
D = vcat(A, B)
```

8.15.8 Transpose

`transpose(a::fmpz_mat)`

Return the transpose of the matrix a.

Examples.

Here is an example of transposing a matrix.

```
S = MatrixSpace(ZZ, 3, 3)

A = S([fmpz(2) 3 5; 1 4 7; 9 6 3])

B = transpose(A)
```

8.15.9 scaling

`<<(a::fmpz_mat, n::Int)`

Return the matrix whose entries are those of a multiplied by 2^n.

`>>(a::fmpz_mat, n::Int)`

Return the matrix whose entries are those of a divided by 2^n, where any remainders are discarded. Rounding is towards zero.

Examples.

Here are some examples of scaling.

```
S = MatrixSpace(ZZ, 3, 3)

A = S([fmpz(2) 3 5; 1 4 7; 9 6 3])

B = A<<5
C = B>>2
```

8.15.10 Exact division

`divexact(a::fmpz_mat, b::fmpz_mat)`

Return a times the inverse of b. The matrix b must be square and invertible, otherwise an exception is raised.

Examples.

Here is an example of exact division.

```
S = MatrixSpace(ZZ, 3, 3)

A = S([fmpz(2) 3 5; 1 4 7; 9 2 2])
B = S([2 3 4; 7 9 1; 5 4 5])

divexact(B, A)
```

8.15.11 Modular reduction

```
reduce_mod(a::fmpz_mat, m::Int)
reduce_mod(a::fmpz_mat, m::fmpz)
```

Return the matrix whose entries are those of a reduced modulo m.

Examples.

Here is an example of modular reduction.

```
S = MatrixSpace(ZZ, 3, 3)

A = S([fmpz(2) 3 5; 1 4 7; 9 2 2])

reduce_mod(A, 5)
reduce_mod(A, 2)
```

8.15.12 Hadamard matrix

```
hadamard(S::FmpzMatSpace)
```

Return the Hadamard matrix for the given matrix space. The number of rows must equal the number of columns. It is not always possible to create a Hadamard matrix. If the algorithm fails, an exception is raised.

```
is_hadamard(a::fmpz_mat)
```

Return true if the matrix a is a Hadamard matrix. The matrix must be square or an exception is raised.

Examples.

Here is an example of computing the Hadamard matrix.

```
S = MatrixSpace(ZZ, 3, 3)

A = hadamard(S)
is_hadamard(A) == true
```

8.15.13 Nullspace

```
nullspace(a::fmpz_mat)
```

Return a tuple (N, n) consisting of the nullity n and a matrix N with c rows and c columns, where c is the number of columns of a, such that the first n columns of N are a basis for the nullspace of a.

Examples.

Here is an example of computing the nullspace.

```
S = MatrixSpace(ZZ, 3, 3)

A = S([fmpz(2) 3 5; 1 4 7; 4 1 1])

N, r = nullspace(A)
```

8.15.14 Hermite normal form

`hnf(a::fmpz_mat)`
Compute the Hermite normal form of a.

`hnf_with_transform(a::fmpz_mat)`

Compute a tuple (H, T) where H is the Hermite normal form of a and T is a transformation matrix so that $H = TA$.

`hnf_modular(a::fmpz_mat, m::fmpz)`

Compute the Hermite normal form of a given that m is a multiple of the determinant of the nonzero rows of a.

`is_hnf(a::fmpz_mat)`

Return `true` if the matrix a is in Hermite normal form, otherwise return false.

Examples.

Here are some examples of computing the Hermite normal form.

```
S = MatrixSpace(ZZ, 3, 3)

A = S([fmpz(2) 3 5; 1 4 7; 19 3 7])

B = hnf(A)
H, T = hnf_with_transform(A)
M = hnf_modular(A, fmpz(27))
is_hnf(M) == true
```

8.15.15 LLL reduction

`lll(a::fmpz_mat)`
Compute the LLL reduction of a.

`lll_with_transform(a::fmpz_mat)`

Compute a tuple (L, T) where L is the LLL reduction of a and T is a transformation matrix so that $L = TA$.

`lll_gram(g::fmpz_mat)`

Given the Gram matrix g of a matrix a, compute the Gram matrix of the LLL reduction of a.

`lll_gram_with_transform(a::fmpz_mat)`

Given the Gram matrix g of a matrix a, compute a tuple (L, T) where G is the gram matrix L of the LLL reduction of a and T is a transformation matrix so that $L = TA$.

`lll_with_removal(a::fmpz_mat, b::fmpz)`

Compute the LLL reduction of a and throw away rows whose norm exceeds the given bound b. Return a tuple (r, L) where the first r rows of L are the rows remaining after removal.

`lll_with_removal_transform(a::fmpz_mat, b::fmpz)`

Compute a tuple (r, L, T) where the first r rows of L are those remaining from the LLL reduction after removal of vectors with norm exceeding the bound b and T is a transformation matrix so that $L = TA$.

Examples.

Here are some examples of computing the LLL reduction.

```
S = MatrixSpace(ZZ, 3, 3)

A = S([fmpz(2) 3 5; 1 4 7; 19 3 7])

L = lll(A)
L, T = lll_with_transform(A)

G == lll_gram(gram(A))
G, T = lll_gram_with_transform(gram(A))

r, L = lll_with_removal(A, fmpz(100))
r, L, T = lll_with_removal_transform(A, fmpz(100))
```

8.15.16 Smith normal form

`snf(a::fmpz_mat)`

Compute the Smith normal form of a.

`is_snf(a::fmpz_mat)`

Return `true` if a is in Smith normal form, otherwise return `false`.

`snf_diagonal(a::fmpz_mat)`

Given a diagonal matrix a compute the Smith normal form of a.

Examples.

Here are some examples of computing the Smith normal form.

```
S = MatrixSpace(ZZ, 3, 3)

A = S([fmpz(2) 3 5; 1 4 7; 19 3 7])

B = snf(A)
is_snf(B) == true

B = S([fmpz(2) 0 0; 0 4 0; 0 0 7])

C = snf_diagonal(B)
```

9 Flint fields

Flint provides implementations of various explicit fields. We describe their Nemo wrappers here.

9.1 Flint rationals ($\mathbb{Q}$): fmpq

Flint provides the module `fmpq` for rational numbers. The type of Flint rationals in Nemo is also given the name `fmpq`.

The parent object for Flint rationals is of type `FlintRationalField`, which belongs to the `Field` type class. Rational number objects have type `fmpq` which belongs to the `FractionElem` type class, which in turn belongs to `FieldElem`.

The parent object for the rationals can be constructed using the `FractionField` function. For example:

```
R = FractionField(ZZ)
```

By default, Nemo makes `QQ = FractionField(ZZ)`, though the user can change this assignment.

The `fmpq` type supports all of the functionality of the generic fraction field section. In this section we describe additional functionality that is provided for Flint rationals but not for general fraction fields.

9.1.1 Constructors

In the constructors below we assume R is a Flint rational field parent object.

```
R(a::Rational{BigInt})
```

Construct an `fmpq` from a Julia big rational a.

One can also construct Flint rationals directly using the `fmpq` constructors.

Examples.

Here is an example of some constructors.

```
R = FractionField(ZZ)

a = R(BigInt(1)//2)
b = fmpq(BigInt(1)//2)
```

9.1.2 Conversions

We provide the means to convert Flint `fmpz` and `fmpq` values to Julia rationals.

```
Rational(a::fmpq)
Rational(a::fmpz)
```

Return a as a Julia big rational.

Examples.

Here is an example of some conversions.

```
a = Rational(fmpz(12))
b = Rational(fmpq(3, 7))
```

9.1.3 Basic manipulation

```
abs(a::fmpq)
```

Return the absolute value of the rational number a.

```
height(a::fmpq)
```

Return the height of the fraction a, namely the largest of the absolute values of the numerator and denominator. The type of the return value is a `ZZ`.

```
height_bits(a::fmpq)
```

Return the number of bits of the height of the fraction a. The type of the return value is an `Int`.

Examples.

Here are some examples of basic manipulation.

```
a = -fmpz(2)//3
b = fmpz(123)//234

c = abs(a)
d = height(a)
e = height_bits(b)
```

9.1.4 Comparison

```
isless(a::fmpq, b::fmpq)
```

Return `true` if $a < b$, otherwise return `false`. Julia supplies operators $<$, $>$, $\leq$ and $\geq$ from this automatically.

Examples.

Here are some examples of comparison.

```
a = -fmpz(2)//3
b = fmpz(1)//2

a < b
b >= a
```

9.1.5 Ad hoc comparison

The following ad hoc comparison operators are provided for convenience.

```
==(a::fmpq, b::Int)
==(a::Int, b::fmpq)
```

Return `true` if a is equal to b, otherwise return `false`. Julia automatically provides `!=` from this.

```
isless(a::fmpq, b::fmpz)
isless(a::fmpq, b::Integer)
isless(a::fmpz, b::fmpq)
isless(a::Integer, b::fmpq)
```

Return `true` if $a < b$, otherwise return `false`. Julia automatically provides corresponding functions `>`, `<=` and `>=`.

Examples.

Here are some examples of ad hoc comparison.

```
a = -fmpz(2)//3
b = fmpz(1)//2

a < 1
a <= 0
b >= fmpz(0)
```

9.1.6 Shifting

```
<<(a::fmpq, b::Int)
```

Return $a \times 2^b$.

```
>>(a::fmpq, b::Int)
```

Return $a/2^b$.

Examples.

Here are some examples of ad hoc binary operators.

```
a = -fmpz(2)//3
b = fmpq(1, 2)

c = a << 3
d = b >> 5
```

9.1.7 Modular arithmetic

```
mod(a::fmpq, m::Integer)
mod(a::fmpq, m::fmpz)
```

If $a = p/q$ this function returns $pq^{-1} \pmod{m}$. This is only defined if q is invertible modulo m. The result is returned as an `fmpz`.

Examples.

Here are some examples of modular arithmetic.

```
a = -fmpz(2)//3
b = fmpz(1)//2

c = mod(a, 7)
d = mod(b, fmpz(5))
```

9.1.8 GCD

Nemo defines the greatest common divisor of two rationals as the greatest rational r such that both the rationals are integer multiples of r.

```
gcd(a::fmpq, b::fmpq)
```

Return the greatest common divisor of a and b.

9.1.9 Rational reconstruction

```
reconstruct(a::fmpz, m::fmpz)
reconstruct(a::fmpz, m::Integer)
reconstruct(a::Integer, m::fmpz)
reconstruct(a::Integer, m::Integer)
```

Attempt to find a rational number n/d such that $0 \leq |n| \leq \lfloor\sqrt{m/2}\rfloor$ and $0 < d \leq \lfloor\sqrt{m/2}\rfloor$ such that $\gcd(n, d) = 1$ and $a \equiv nd^{-1} \pmod{m}$. If no solution exists, an exception is thrown.

Examples.

Here are some examples of rational reconstruction.

```
a = reconstruct(7, 13)
b = reconstruct(fmpz(15), 31)
c = reconstruct(fmpz(123), fmpz(237))
```

9.1.10 Rational enumeration

```
next_minimal(x::fmpq)
```

Given x, returns the next rational number in the sequence obtained by enumerating all positive denominators q, and for each q enumerating the numerators $1 \le p < q$ in order and generating both p/q and q/p, but skipping all $\gcd(p, q) \neq 1$. Starting with zero, this generates every nonnegative rational number once and only once, with the first few entries being:

$$0, 1, 1/2, 2, 1/3, 3, 2/3, 3/2, 1/4, 4, 3/4, 4/3, 1/5, 5, 2/5, \ldots.$$

This enumeration produces the rational numbers in order of minimal height. It has the disadvantage of being somewhat slower to compute than the Calkin-Wilf enumeration.

If $x < 0$ we throw a `DomainError()`.

`next_signed_minimal(x::fmpq)`

Given a signed rational number x assumed to be in canonical form, returns the next element in the minimal-height sequence generated by `fmpq_next_minimal` but with negative numbers interleaved:

$$0, 1, -1, 1/2, -1/2, 2, -2, 1/3, -1/3, \ldots.$$

Starting with zero, this generates every rational number once and only once, in order of minimal height.

`next_calkin_wilf(x::fmpq)`

Given x return the next number in the breadth-first traversal of the Calkin-Wilf tree. Starting with zero, this generates every nonnegative rational number once and only once, with the first few entries being:

$$0, 1, 1/2, 2, 1/3, 3/2, 2/3, 3, 1/4, 4/3, 3/5, 5/2, 2/5, \ldots.$$

Despite the appearance of the initial entries, the Calkin-Wilf enumeration does not produce the rational numbers in order of height: some small fractions will appear late in the sequence. This order has the advantage of being faster to produce than the minimal-height order.

`next_signed_calkin_wilf(x::fmpq)`

Given a signed rational number x returns the next element in the Calkin-Wilf sequence with negative numbers interleaved:

$$0, 1, -1, 1/2, -1/2, 2, -2, 1/3, -1/3, \ldots.$$

Starting with zero, this generates every rational number once and only once, but not in order of minimal height.

Examples.

Here are some examples of rational enumeration.

```
next_minimal(fmpz(2)//3)
next_signed_minimal(-fmpz(21)//31)
next_calkin_wilf(fmpz(321)//113)
next_signed_calkin_wilf(-fmpz(51)//(17))
```

9.1.11 Special functions

`harmonic(n::Int)`

Computes the harmonic number $H_n = 1 + 1/2 + 1/3 + \cdots + 1/n$. Table lookup is used for H_n whose numerator and denominator fit in a single limb. For larger n, a divide and conquer strategy is used.

`bernoulli(n::Int)`

Computes the Bernoulli number B_n for nonnegative n.

`bernoulli_cache(n::Int)`

Precomputes and caches all the Bernoulli numbers up to B_n. This is much faster than repeatedly calling `bernoulli(k)`. Once cached, subsequent calls to `bernoulli(k)` for any $k \leq n$ will read from the cache, making them virtually free.

```
dedekind_sum(h::fmpz, k::fmpz)
dedekind_sum(h::fmpz, k::Integer)
dedekind_sum(h::Integer, k::fmpz)
dedekind_sum(h::Integer, k::Integer)
```

Computes the Dedekind sum $s(h, k)$ for arbitrary h and k.

Examples.

Here are some examples of special functions.

```
a = harmonic(12)

b = dedekind_sum(12, 13)
c = dedekind_sum(-120, fmpz(1305))

d = bernoulli(12)

bernoulli_cache(100)
e = bernoulli(100)
```

9.2 Flint finite fields $\mathbb{F}_{p^k}$ (multiprecision p): `fq`

We allow the construction of finite fields of any characteristic and degree in Nemo. When a Conway polynomial is known, the field is generated using the Conway polynomial. Otherwise a random sparse, irreducible polynomial is used.

(At this stage we make no attempt to compatibly embed finite fields generated using randomly chosen irreducible polynomials, such as with the Bosma-Cannon-Steel construction. But this may change in a later version of Flint/Nemo.)

The Flint type for finite fields $\mathbb{F}_{p^k}$ for multiprecision p is `fq`.

The type of elements of Flint `fq` fields in Nemo is `fq`. This type belongs to the `FiniteFieldElem` type class, which in turn belongs to the `FieldElem` class. The type of the parent object for such fields is `FqFiniteField` which belongs to the `Field` type class.

The parent object for such a field is created using the `FiniteField` function, e.g.

```
FiniteField(p::fmpz, deg::Int, var::String)
```

> Return a tuple `(R, x)` consisting of the parent object `R` of a finite field of characteristic `p` of degree `deg`, and a generator `x`.
>
> The generator `x` of the field will be printed as the provided string `var`. The characteristic `p` must be prime, but we do not check this condition. The degree `deg` must be nonnegative. If `deg` is negative we throw a `DomainError()`.

For example, the following code generates a parent object for the finite field of characteristic 7 and degree 5 (defined by a Conway polynomial).

```
S, x = FiniteField(fmpz(7), 5, "x")
```

9.2.1 Constructors

Elements of the finite field `R` can be constructed as polynomials of degree less than `deg` in `x`. As usual we overload the parent object for the finite field to construct elements of the field.

In the constructors below, we take S to be a parent object of a finite field as given by the construction above, for example.

```
S()
```

> Return the element 0 of the finite field.

```
S(a::Int)
S(a::Integer)
S(a::fmpz)
```

> Return the element `a` of the finite field, where a is considered an element of $\mathbb{F}_p$.

```
S(a::fq)
```

> Return a reference to the element `a` of the finite field. No copy of the data is made by this function.

Examples.

Here are some examples of constructing finite fields and elements in them.

```
S, y = FiniteField(fmpz(17), 3, "y")

f = 2y^2 + 11y + 16

g = S(4)
h = S()
k = S(g)
```

9.2.2 Basic manipulation

`zero(R::FqFiniteField)`
Return the additive identity in the finite field, i.e. 0.

`one(R::FqFiniteField)`

Return the multiplicative identity in the finite field, i.e. 1.

`gen(R::FqFiniteField)`

Return a generator of the finite field, namely one whose representation is the monomial of degree 1 modulo the defining polynomial of the field. The generator is guaranteed to be a multiplicative generator only if the field is generated by a Conway polynomial.

`iszero(a::fq)`

Return `true` if the element a is the additive identity in the finite field, i.e. if it is 0, otherwise return `false`.

`isone(a::fq)`

Return `true` if the element a is the multiplicative identity in the finite field, i.e. if it is 1, otherwise return `false`.

`isgen(a::fq)`

Return `true` if the element a is the generator of the finite field, otherwise return `false`.

`isunit(a::fq)`

Return `true` if the element a is invertible in the finite field, i.e. if it is not zero, otherwise return `false`.

`characteristic(R::FqFiniteField)`

Return the prime characteristic p of the finite field as a bignum `ZZ`.

`order(R::FqFiniteField)`

Return the number of elements in the finite field, i.e. $q = p^d$ where `p` is the characteristic and d is the degree.

`degree(R::FqFiniteField)`

If the finite field is $\mathbb{F}_q$ where $q = p^d$, where p is the prime characteristic of the field, this function returns the degree d.

```
coeff(a::fq, n::Int)
```

Return the coefficient of degree n in the polynomial representation of the element a of the finite field. If $n < 0$ we throw a `DomainError()`.

```
deepcopy(a::fq)
```

Return a new finite field element which is arithmetically equal to the given element.

```
canonical_unit(a::fq)
```

Used for canonicalising fractions. Simply returns a.

Examples.

Here are some examples of basic manipulations of finite fields.

```
R, x = FiniteField(fmpz(7), 5, "x")

a = zero(R)
b = one(R)
c = gen(R)
d = characteristic(R)
f = order(R)
g = degree(R)
h = iszero(a)
k = isone(b)
m = isunit(x + 1)
n = deepcopy(c)
```

9.2.3 Unary operators

```
-(a::fq)
```

Return $-a$.

Examples.

Here are some examples of unary operators.

```
R, x = FiniteField(fmpz(7), 5, "x")

a = x^4 + 3x^2 + 6x + 1
b = -a
```

9.2.4 Binary operators

We provide the following binary operators for elements of finite fields.

```
+(a::fq, b::fq)
```

Return $a + b$.

```
-(a::fq, b::fq)
```

Return $a - b$.

```
*(a::fq, b::fq)
```

Return ab.

Examples.

Here are some examples of binary operators.

```
R, x = FiniteField(fmpz(7), 5, "x")

a = x^4 + 3x^2 + 6x + 1
b = 3x^4 + 2x^2 + x + 1

c = a + b
d = a - b
f = a*b
```

9.2.5 Ad hoc binary operators

We provide the following ad hoc binary operators, which are faster than first coercing all their arguments into the finite field.

```
*(x::Int, y::fq)
*(x::Integer, y::fq)
*(x::fmpz, y::fq)
```

Return xy, i.e. y added to itself x times.

```
*(x::fq, y::Int)
*(x::fq, y::Integer)
*(x::fq, y::fmpz)
```

Return xy, i.e. x added to itself y times.

Examples.

Here are some examples of ad hoc binary operators.

```
R, x = FiniteField(fmpz(7), 5, "x")

a = x^4 + 3x^2 + 6x + 1

b = 3a
c = a*fmpz(5)
```

9.2.6 Powering

```
^(a::fq, n::Int)
^(a::fq, n::fmpz)
```

Return a^n.

Examples.

Here are some examples of powering of finite field elements.

```
R, x = FiniteField(fmpz(7), 5, "x")

a = x^4 + 3x^2 + 6x + 1

b = a^3
c = a^fmpz(-5)
```

9.2.7 Comparison

```
==(a::fq, b::fq)
```

Return `true` if the finite field elements a and b are arithmetically equal, otherwise return `false`.

Julia automatically provides corresponding `!=` functionality.

Examples.

Here are some examples of comparison of finite field elements.

```
R, x = FiniteField(fmpz(7), 5, "x")

a = x^4 + 3x^2 + 6x + 1
b = 3x^4 + 2x^2 + 2

b != a
R(3) == R(3)
```

9.2.8 Inversion

```
inv(a::fq)
```

Return the multiplicative inverse of a in the finite field, i.e. a^{-1} such that $aa^{-1} = 1$ in the finite field.

Examples.

Here are some examples of inversion of finite field elements.

```
R, x = FiniteField(fmpz(7), 5, "x")

a = x^4 + 3x^2 + 6x + 1

b = inv(a)
b == a^-1
```

9.2.9 Exact division

```
divexact(x::fq, y::fq)
//(x::fq, y::fq)
```

Return x/y, which is an exact division in a finite field when defined, since every nonzero element is invertible. We throw a `DivideError()` if $y = 0$. Both forms of exact division are the same, one being a synonym for the other.

Examples.

Here are some examples of exact division of finite field elements.

```
R, x = FiniteField(fmpz(7), 5, "x")

a = x^4 + 3x^2 + 6x + 1
b = 3x^4 + 2x^2 + 2

c = divexact(a, b)
d = b//a
```

9.2.10 GCD

```
gcd(a::fq, b::fq)
```

Return $\gcd(a, b)$. For a finite field this is always 1 unless both a and b are 0, in which case the gcd is 0.

Examples.

Here are some examples of GCD.

```
R, x = FiniteField(fmpz(7), 5, "x")

a = x^4 + 3x^2 + 6x + 1
b = 3x^4 + 2x^2 + x + 1

c = gcd(a, b)
d = gcd(R(0), R(0))
```

9.2.11 Special functions

Various special functions with finite field specific behaviour are defined.

```
trace(a::fq)
```

Return the trace of a. This is an element of $\mathbb{F}_p$, but the value returned is this value embedded in the original finite field.

```
norm(a::fq)
```

Return the norm of a. This is an element of $\mathbb{F}_p$, but the value returned is this value embedded in the original finite field.

```
frobenius(a::fq, n = 1)
```

Return the iterated Frobenius $\sigma_p^n(a)$ where σ_p is the Frobenius map sending the element a to a^p in the finite field of characteristic p. By default the Frobenius map is applied $n = 1$ times if n is not specified.

```
pth_root(a::fq)
```

Return the p-th root of a in the finite field of characteristic p. This is the inverse operation to the Frobenius map σ_p.

Examples.

Here are some examples of special finite field functionality.

```
R, x = FiniteField(fmpz(7), 5, "x")

a = x^4 + 3x^2 + 6x + 1

b = trace(a)
c = norm(a)
d = frobenius(a)
f = frobenius(a, 3)
g = pth_root(a)
```

9.3 Flint finite fields $\mathbb{F}_{p^k}$ (small p): fq_nmod

The Flint type for finite fields $\mathbb{F}_{p^k}$ for p that fits in a single machine word is fq_nmod.

When a Conway polynomial is known, such a field is generated using the Conway polynomial. Otherwise a random sparse, irreducible polynomial is used.

The type of elements of Flint fq_nmod fields in Nemo is fq_nmod. This type belongs to the FiniteFieldElem type class, which in turn belongs to the FieldElem class. The type of the parent object for such fields is FqNmodFiniteField which belongs to the Field type class.

The parent object for such a field is created using the FiniteField function, e.g.

```
FiniteField(p::Int, deg::Int, var::String)
```

Return a tuple (R, x) consisting of the parent object R of a finite field of characteristic p of degree deg, and a generator x.

The generator x of the field will be printed as the provided string var. The characteristic p must be prime, but we do not check this condition. The degree deg must be nonnegative. If deg is negative we throw a DomainError().

For example, the following code generates a parent object for the finite field of characteristic 7 and degree 5 (defined by a Conway polynomial).

```
S, x = FiniteField(7, 5, "x")
```

9.3.1 Constructors

Elements of the finite field `R` can be constructed as polynomials of degree less than `deg` in `x`. As usual we overload the parent object for the finite field to construct elements of the field.

In the constructors below, we take S to be a parent object of a finite field as given by the construction above, for example.

`S()`

Return the element 0 of the finite field.

`S(a::Int)`
`S(a::Integer)`
`S(a::fmpz)`

Return the element `a` of the finite field, where a is considered an element of $\mathbb{F}_p$.

`S(a::fq_nmod)`

Return a reference to the element `a` of the finite field. No copy of the data is made by this function.

Examples.

Here are some examples of constructing finite fields and elements in them.

```
S, y = FiniteField(17, 3, "y")

f = 2y^2 + 11y + 16

g = S(4)
h = S()
k = S(g)
```

9.3.2 Basic manipulation

`zero(R::FqNmodFiniteField)`
Return the additive identity in the finite field, i.e. 0.

`one(R::FqNmodFiniteField)`

Return the multiplicative identity in the finite field, i.e. 1.

`gen(R::FqNmodFiniteField)`

Return a generator of the finite field, namely one whose representation is the monomial of degree 1 modulo the defining polynomial of the field. The generator is guaranteed to be a multiplicative generator only if the field is generated by a Conway polynomial.

`iszero(a::fq_nmod)`

Return `true` if the element a is the additive identity in the finite field, i.e. if it is 0, otherwise return `false`.

`isone(a::fq_nmod)`

Return `true` if the element a is the multiplicative identity in the finite field, i.e. if it is 1, otherwise return `false`.

`isgen(a::fq_nmod)`

Return `true` if the element a is the generator of the finite field, otherwise return `false`.

`isunit(a::fq_nmod)`

Return `true` if the element a is invertible in the finite field, i.e. if it is not zero, otherwise return `false`.

`characteristic(R::FqNmodFiniteField)`

Return the prime characteristic p of the finite field as a bignum `ZZ`.

`order(R::FqNmodFiniteField)`

Return the number of elements in the finite field, i.e. $q = p^d$ where `p` is the characteristic and d is the degree.

`degree(R::FqNmodFiniteField)`

If the finite field is $\mathbb{F}_q$ where $q = p^d$, where p is the prime characteristic of the field, this function returns the degree d.

`coeff(a::fq_nmod, n::Int)`

Return the coefficient of degree n in the polynomial representation of the element a of the finite field. If $n < 0$ we throw a `DomainError()`.

`deepcopy(a::fq_nmod)`

Return a new finite field element which is arithmetically equal to the given element.

`canonical_unit(a::fq_nmod)`

Used for canonicalising fractions. Simply returns a.

Examples.

Here are some examples of basic manipulations of finite fields.

```
R, x = FiniteField(7, 5, "x")
a = zero(R)
b = one(R)
c = gen(R)
d = characteristic(R)
f = order(R)
g = degree(R)
h = iszero(a)
k = isone(b)
m = isunit(x + 1)
n = deepcopy(c)
```

9.3.3 Unary operators

```
-(a::fq_nmod)
```

Return $-a$.

Examples.

Here are some examples of unary operators.

```
R, x = FiniteField(7, 5, "x")
a = x^4 + 3x^2 + 6x + 1
b = -a
```

9.3.4 Binary operators

We provide the following binary operators for elements of finite fields.

```
+(a::fq_nmod, b::fq_nmod)
```

Return $a + b$.

```
-(a::fq_nmod, b::fq_nmod)
```

Return $a - b$.

```
*(a::fq_nmod, b::fq_nmod)
```

Return ab.

Examples.

Here are some examples of binary operators.

```
R, x = FiniteField(7, 5, "x")
a = x^4 + 3x^2 + 6x + 1
b = 3x^4 + 2x^2 + x + 1

c = a + b
d = a - b
f = a*b
```

9.3.5 Ad hoc binary operators

We provide the following ad hoc binary operators, which are faster than first coercing all their arguments into the finite field.

```
*(x::Int, y::fq_nmod)
*(x::Integer, y::fq_nmod)
*(x::fmpz, y::fq_nmod)
```

Return xy, i.e. y added to itself x times.

```
*(x::fq_nmod, y::Int)
*(x::fq_nmod, y::Integer)
*(x::fq_nmod, y::fmpz)
```

Return xy, i.e. x added to itself y times.

Examples.

Here are some examples of ad hoc binary operators.

```
R, x = FiniteField(7, 5, "x")
a = x^4 + 3x^2 + 6x + 1

b = 3a
c = a*fmpz(5)
```

9.3.6 Powering

```
^(a::fq_nmod, n::Int)
^(a::fq_nmod, n::fmpz)
```

Return a^n.

Examples.

Here are some examples of powering of finite field elements.

```
R, x = FiniteField(7, 5, "x")
a = x^4 + 3x^2 + 6x + 1

b = a^3
c = a^fmpz(-5)
```

9.3.7 Comparison

```
==(a::fq_nmod, b::fq_nmod)
```

Return `true` if the finite field elements a and b are arithmetically equal, otherwise return `false`.

Julia automatically provides corresponding `!=` functionality.

Examples.

Here are some examples of comparison of finite field elements.

```
R, x = FiniteField(7, 5, "x")
a = x^4 + 3x^2 + 6x + 1
b = 3x^4 + 2x^2 + 2

b != a
R(3) == R(3)
```

9.3.8 Inversion

`inv(a::fq_nmod)`
Return the multiplicative inverse of a in the finite field, i.e. a^{-1} such that $aa^{-1} = 1$ in the finite field.

Examples.

Here are some examples of inversion of finite field elements.

```
R, x = FiniteField(7, 5, "x")
a = x^4 + 3x^2 + 6x + 1

b = inv(a)
b == a^-1
```

9.3.9 Exact division

`divexact(x::fq_nmod, y::fq_nmod)`
`//(x::fq_nmod, y::fq_nmod)`
Return x/y, which is an exact division in a finite field when defined, since every nonzero element is invertible. We throw a `DivideError()` if $y = 0$. Both forms of exact division are the same, one being a synonym for the other.

Examples.

Here are some examples of exact division of finite field elements.

```
R, x = FiniteField(7, 5, "x")
a = x^4 + 3x^2 + 6x + 1
b = 3x^4 + 2x^2 + 2

c = divexact(a, b)
d = b//a
```

9.3.10 GCD

`gcd(a::fq_nmod, b::fq_nmod)`
Return $\gcd(a, b)$. For a finite field this is always 1 unless both a and b are 0, in which case the gcd is 0.

Examples.

Here are some examples of GCD.

```
R, x = FiniteField(7, 5, "x")
a = x^4 + 3x^2 + 6x + 1
b = 3x^4 + 2x^2 + x + 1

c = gcd(a, b)
d = gcd(R(0), R(0))
```

9.3.11 Special functions

Various special functions with finite field specific behaviour are defined.

`trace(a::fq_nmod)`

Return the trace of a. This is an element of $\mathbb{F}_p$, but the value returned is this value embedded in the original finite field.

`norm(a::fq_nmod)`

Return the norm of a. This is an element of $\mathbb{F}_p$, but the value returned is this value embedded in the original finite field.

`frobenius(a::fq_nmod, n = 1)`

Return the iterated Frobenius $\sigma_p^n(a)$ where σ_p is the Frobenius map sending the element a to a^p in the finite field of characteristic p. By default the Frobenius map is applied $n = 1$ times if n is not specified.

`pth_root(a::fq_nmod)`

Return the p-th root of a in the finite field of characteristic p. This is the inverse operation to the Frobenius map σ_p.

Examples.

Here are some examples of special finite field functionality.

```
R, x = FiniteField(7, 5, "x")
a = x^4 + 3x^2 + 6x + 1

b = trace(a)
c = norm(a)
d = frobenius(a)
f = frobenius(a, 3)
g = pth_root(a)
```

9.4 Flint p-adics $\mathbb{Q}_p$: `padic`

The Flint type for p-adic numbers is `padic`. In Nemo elements of a p-adic field are of type `padic`. These belong to a `PadicFieldElem` type class, which in turn belongs to `FieldElem`. The parent objects for the p-adic fields are of type `FlintPadicField` which belongs to the `Field` type class.

As usual, the user need not deal directly with the types, as we define a function `FlintPadicField` for constructing parent objects for p-adic fields.

By default, the global variable `PadicField` is defined to be `FlintPadicField` so that `PadicField` also constructs parent objects for p-adic fields.

The p-adics implement a capped relative precision model. This has consistent behaviour for multiplicative operations on p-adics, but not for additive operations, where cancellation can occur.

9.4.1 Constructors

We provide the following simple function for creating the type of the p-adic numbers.

```
PadicField(p::Int, prec::Int)
PadicField(p::ZZ, prec::Int)
```

> This function returns a type corresponding to the p-adic numbers for the given prime p. We test that p is actually prime before constructing the type. If not, an exception is thrown. The second argument is a default (absolute) precision for elements in this field.

To construct values of p-adic type we use the `O` operator. It requires the p-adic type as a parameter for reasons of type soundness and so that expressions such as $O(7^0)$ can be entered unambiguously (as `O(R, 7^0)`, say).

```
O(R::FlintPadicField, m::Int)
O(R::FlintPadicField m::fmpz)
O(R::FlintPadicField, m::fmpq)
```

> Construct the value $0 + O(p^n)$ given $m = p^n$. An exception results if m is not found to be a power of `p = prime(R)`.

The `O(p^n)` construction can be used to construct p-adic values of precision n by adding it to integer values representing the p-adic value modulo p^n. See the examples below for details.

In the constructors below, we assume S is a p-adic field parent object, constructed using the `PadicField` function, for example:

```
S = PadicField(17, 30)
```

We also assume below that p is the prime passed to the `PadicField` constructor, e.g. $p = 17$ in the above and that k is the default precision, e.g. $k = 30$ in the above.

```
S()
```

> Construct the p-adic value 0 with default precision, i.e. $0 + O(p^k)$.

```
S(a::Int)
S(a::fmpz)
S(a::fmpq)
```

> Construct the p-adic value representing the integer or rational a to precision k.

Examples.

Here are some examples of constructing p-adics.

```
R = PadicField(7, 30)
S = PadicField(fmpz(65537), 30)

a = R()
b = R(1)
c = R(fmpz(123))
d = R(fmpz(1)//7^2)

f = 1 + 2*7 + 4*7^2 + O(R, 7^3)
g = 13 + 357*fmpz(65537) + O(S, fmpz(65537)^12)
h = fmpz(1)//7^2 + fmpz(2)//7 + 3 + 4*7 + O(R, 7^2)
```

Beware that the expression `1 + 2*p + 3*p^2 + O(R, p^n)` is actually computed as a normal Julia expression. Therefore if `Int` values are used instead of `fmpz`'s, overflow may result in evaluating the value.

Also note that one cannot use `7^-2` in a p-adic expression, since the exponentiation operator must always return an integer, for reasons of type soundness.

9.4.2 Basic manipulation

`prime(R::PadicField)`

Return the prime p on which the p-adic type is based. The returned value is of type `fmpz`.

`precision(a::padic)`

Return the precision of the given p-adic value, e.g. the precision of $1 + 7 + 2 * 7^2 + O(7^3)$ is 3.

`valuation(a::padic)`

Return the p-adic valuation of the given p-adic value.

`zero(R::PadicField)`

Return the additive identity, i.e. 0 in the field of p-adic numbers to the default precision.

`one(R::PadicField)`

Return the multiplicative identity, i.e. 1 in the field of p-adic numbers to the default precision.

`iszero(a::padic)`

Return `true` if the given p-adic value is equal to the additive identity, i.e. 0, otherwise return `false`.

`isone(a::padic)`

Return `true` if the given p-adic value is equal to the multiplicative identity, i.e. 1, otherwise return `false`.

Examples.

Here are some examples of basic manipulations of p-adics.

```
R = PadicField(7, 30)

a = 1 + 2*7 + 4*7^2 + O(R, 7^3)
b = 7^2 + 3*7^3 + O(R, 7^5)
c = R(2)

d = one(R)
f = zero(R)
g = isone(d)
h = iszero(f)
k = precision(a)
m = prime(R)
n = valuation(b)
```

9.4.3 Unary operators

`-(x::padic)`
Return $-x$.

Examples.

Here are some examples of unary operators.

```
R = PadicField(7, 30)

a = 1 + 2*7 + 4*7^2 + O(R, 7^3)
b = R(0)

c = -a
d = -b
```

9.4.4 Binary operators

`+(x::padic, y::padic)`
Return $x + y$. The output precision will be the least of the input precisions.

`-(x::padic, y::padic)`

Return $x - y$. The output precision will be the least of the input precisions.

`*(x::padic, y::padic)`

Return xy. The output precision will be the least of valuation(x)+ precision(y) and precision(x)+ valuation(y).

Examples.

Here are some examples of binary operators for p-adics.

```
R = PadicField(7, 30)

a = 1 + 2*7 + 4*7^2 + O(R, 7^3)
b = 7^2 + 3*7^3 + O(R, 7^5)
c = O(R, 7^3)
d = R(2)

f = a + b
g = a - b
h = a*b
j = b*c
k = a*d
```

9.4.5 Ad hoc binary operators

```
+(x::padic, y::Int)
+(x::padic, y::fmpz)
+(x::padic, y::fmpq)
```

Return $x + y$ where y is interpreted as a p-adic value to the default precision.

```
+(x::Int, y::padic)
+(x::fmpz, y::padic)
+(x::fmpq, y::padic)
```

Return $x + y$ where x is interpreted as a p-adic value to the default precision.

```
-(x::padic, y::Int)
-(x::padic, y::fmpz)
-(x::padic, y::fmpq)
```

Return $x - y$ where y is interpreted as a p-adic value to the default precision.

```
-(x::Int, y::padic)
-(x::fmpz, y::padic)
-(x::fmpq, y::padic)
```

Return $x - y$ where x is interpreted as a p-adic value to the default precision.

```
*(x::padic, y::Int)
*(x::padic, y::fmpz)
*(x::padic, y::fmpq)
```

Return xy.

```
*(x::Int, y::padic)
*(x::fmpz, y::padic)
*(x::fmpq, y::padic)
```

Return xy.

Examples.

Here are some examples of ad hoc binary operators.

```
R = PadicField(7, 30)

a = 1 + 2*7 + 4*7^2 + O(R, 7^3)
b = 7^2 + 3*7^3 + O(R, 7^5)
c = O(R, 7^3)
d = R(2)

f = a + 2
g = 3 - b
```

```
h = a*fmpz(5)
j = fmpz(3)*c
k = 2*d
l = 2 + d
m = d - fmpz(2)
n = a + fmpz(1)//7^2
p = (fmpz(12)//11)*b
q = c*(fmpz(1)//7)
```

9.4.6 Comparison

```
==(a::padic, b::padic)
```

Compare the given p-adic values at the least of the two precisions and return `true` if they are equal to that precision, otherwise return `false`.

Julia automatically supplies a corresponding `!=` operator.

```
isequal(a::padic, b::padic)
```

Return `true` if `a == b` and their precisions are the same, otherwise return `false`.

Examples.

Here are some examples of comparison.

```
R = PadicField(7, 30)

a = 1 + 2*7 + 4*7^2 + O(R, 7^3)
b = 3*7^3 + O(R, 7^5)
c = O(R, 7^3)
d = R(2)

a == 1 + 2*7 + O(R, 7^2)
b == c
c == R(0)
d == R(2)
isequal(b, c) == false
```

9.4.7 Ad hoc comparison

The following ad hoc comparison operators are provided for convenience.

```
==(a::padic, b::Int)
==(a::padic, b::fmpz)
==(a::padic, b::fmpq)
```

Return `true` if the value b is equal to the p-adic value a up to the precision of a, otherwise return `false`.

```
==(a::Int, b::padic)
==(a::fmpz, b::padic)
==(a::fmpq, b::padic)
```

Return `true` if the value a is equal to the p-adic value b up to the precision of b, otherwise return `false`.

Examples.

Here are some examples of ad hoc comparison.

```
R = PadicField(7, 30)

a = 1 + O(R, 7^3)
b = O(R, 7^5)
c = R(2)

a == 1
b == fmpz(0)
c == 2
fmpz(2) == c
a == fmpz(344)//1
```

9.4.8 Powering

```
^(a::padic, n::Int)
```

Return a^n. The output precision will be the same as if a had been multiplied by itself n times, i.e. precision(a) + $(n-1)\times$ valuation(a).

Examples.

Here are some examples of powering.

```
R = PadicField(7, 30)

a = 1 + 7 + 2*7^2 + O(R, 7^3)
b = O(R, 7^5)
c = R(2)

d = a^5
f = b^3
g = c^7
```

9.4.9 Inversion

```
inv(a::padic)
```

Return the multiplicative inverse of a, i.e. $1/a$. If `a == 0` a `DivideError()` is thrown.

Examples.

Here are some examples of inversion.

```
R = PadicField(7, 30)

a = 1 + 7 + 2*7^2 + O(R, 7^3)
b = 2 + 3*7 + O(R, 7^5)
c = 7^2 + 2*7^3 + O(R, 7^4)
d = 7 + 2*7^2 + O(R, 7^5)

f = inv(a)
g = inv(b)
h = inv(c)
k = inv(d)
l = inv(R(1))
```

9.4.10 Exact division

```
divexact(a::padic, b::padic)
//(a::padic, b::padic)
```

Return a/b. The output precision will be the minimum of precision$(a)-$ valuation(b) and precision$(b) - 2\times$valuation$(b)+$ valuation(a). If `b == 0` a `DivideError()` is thrown.

Examples.

Here are some examples of exact division.

```
R = PadicField(7, 30)

a = 1 + 7 + 2*7^2 + O(R, 7^3)
b = 2 + 3*7 + O(R, 7^5)
c = 7^2 + 2*7^3 + O(R, 7^4)
d = 7 + 2*7^2 + O(R, 7^5)

f = divexact(a, b)
g = divexact(c, d)
h = divexact(d, R(7^3))
j = divexact(R(34), R(17))
```

9.4.11 Ad hoc exact division

```
divexact(a::padic, b::Int)
divexact(a::padic, b::fmpz)
divexact(a::padic, b::fmpq)
//(a::padic, b::Int)
//(a::padic, b::fmpz)
//(a::padic, b::fmpq)
```

Return a/b. The output precision will be precision$(a)-$ valuation(b). If `b == 0` a `DivideError()` is thrown.

```
divexact(a::Int, b::padic)
divexact(a::fmpz, b::padic)
divexact(a::fmpq, b::padic)
//(a::Int, b::padic)
//(a::fmpz, b::padic)
//(a::fmpq, b::padic)
```

Return a/b. The output precision will be precision(b) − 2×valuation(b)+ valuation(a). If `b == 0` a `DivideError()` is thrown.

Examples.

Here are some examples of ad hoc exact division.

```
R = PadicField(7, 30)

a = 1 + 7 + 2*7^2 + O(R, 7^3)
b = 2 + 3*7 + O(R, 7^5)
c = 7^2 + 2*7^3 + O(R, 7^4)
d = 7 + 2*7^2 + O(R, 7^5)

f = a//2
g = b//fmpz(7)
h = c//(fmpz(12)//7^2)
k = 2//d
l = R(3)//3
m = (fmpz(5)//7)//R(5)
```

9.4.12 GCD

```
gcd(a::padic, b::padic)
```

Returns 1 unless both a and b are 0, in which case it returns 0.

Examples.

Here are some examples of GCD.

```
R = PadicField(7, 30)

a = 1 + 7 + 2*7^2 + O(R, 7^3)
b = 2 + 3*7 + O(R, 7^5)

d = gcd(a, b)
f = gcd(R(0), R(0))
```

9.4.13 Square root

```
sqrt(a::padic)
```

Return the p-adic square root of a. We define this only when the valuation of a is even. The precision of the output will be precision(a)− valuation$(a)/2$. If the square root does not exist, an exception is thrown.

Examples.

Here are some examples of taking a p-adic square root.

```
R = PadicField(7, 30)

a = 1 + 7 + 2*7^2 + O(R, 7^3)
b = 2 + 3*7 + O(R, 7^5)
c = 7^2 + 2*7^3 + O(R, 7^4)

d = sqrt(a)
f = sqrt(b)
f = sqrt(c)
g = sqrt(R(121))
```

9.4.14 Special functions

`exp(a::padic)`

Return the p-adic exponential of a. We define this only when the valuation of a is positive (unless `a == 0`). The precision of the output will be the same as the precision of the input. If the input is not valid an exception is thrown.

`log(a::padic)`

Return the p-adic logarithm of a. We define this only when the valuation of a is zero (but not for `a == 0`). The precision of the output will be the same as the precision of the input. If the input is not valid an exception is thrown.

`teichmuller(a::padic)`

Return the Teichmuller lift of the p-adic value a. We require the valuation of a to be non-negative. The precision of the output will be the same as the precision of the input. For convenience, if a is congruent to zero modulo p we return zero. If the input is not valid an exception is thrown.

Examples.

Here are some examples of special functions.

```
R = PadicField(7, 30)

a = 1 + 7 + 2*7^2 + O(R, 7^3)
b = 2 + 5*7 + 3*7^2 + O(R, 7^3)
c = 3*7 + 2*7^2 + O(R, 7^5)

c = exp(c)
d = log(a)
c = exp(R(0))
d = log(R(1))
f = teichmuller(b)
```

10 Antic fields

Antic is a library for number fields, built on top of Flint. For now it provides us with number field arithmetic.

10.1 Antic number fields $\mathbb{Q}[x]/(f)$: nf_elem

Number fields in Antic are represented by an irreducible polynomial f over $\mathbb{Q}$, with special code to deal with the case where f happens to be defined over $\mathbb{Z}$ and monic.

Antic has a type nf_elem for elements of a number field.

In Nemo the parent object of a number field has type AnticNumberField which belongs to the Field type class. Elements of a number field are of type nf_elem which belongs to the NumberFieldElem type class, which belongs to codeFieldElem.

The user need not deal with the types directly since we provide a function AnticNumberField to create parent objects for number fields and we also provide various constructors to create elements of a number field.

By default, the global variable NumberField is set equal to AnticNumberField which means that the following code can be used to create number field objects in Nemo.

```
R, x = PolynomialRing(QQ, "x")
K, a = NumberField(x^3 + 3x + 1, "a")
```

Here, $x^3 + 3x + 1$ is the irreducible polynomial over $\mathbb{Q}$ giving the minimum polynomial of the generator a of the number field K (though it need not be a monic polynomial).

10.1.1 Constructors

Various constructors are provided to create elements of a number field in Nemo. In the constructors below we assume that K is the parent object of a number field, for example created by the construction above.

```
K()
```

Create the element 0 of the number field.

```
K(a::Int)
K(a::fmpz)
K(a::fmpq)
```

Create the element of the number field represented by the degree 0 polynomial with constant coefficient a.

```
K(g::fmpq_poly)
```

Create the element of the number field represented by the polynomial g. The polynomial is first reduced modulo the defining polynomial f of the number field, thus they must both belong to the same polynomial ring.

```
K(d::nf_elem)
```

Return a reference to the number field element d. No copy is made of the data.

Examples.

Here are some examples of constructors.

```
R, x = PolynomialRing(QQ, "x")
K, a = NumberField(x^3 + 3x + 1, "a")

k = K()
l = K(123)
m = K(fmpz(12))
n = K(x^2 + 2x - 7)
p = K(n)
```

10.1.2 Conversions

If R is the parent object of the polynomial ring to which the defining polynomial f of the number field belongs, we can coerce the polynomial representation g of a number field element d into the polynomial ring defined by R. This allows us to retrieve the polynomial representation of number field elements as a polynomial.

```
R(d::nf_elem)
```

Return the polynomial representation of the number field element d in the polynomial ring R which was used to define the number field.

Examples.

Here is an example of a conversion.

```
R, x = PolynomialRing(QQ, "x")
K, a = NumberField(x^3 + 3x + 1, "a")

k = K(x^2 + 2x - 7)
l = R(k)
```

10.1.3 Basic manipulation

```
zero(R::AnticNumberField)
```

Return the element 0 of the number field with parent object R.

```
one(R::AnticNumberField)
```

Return the element 1 of the number field with parent object R.

```
gen(R::AnticNumberField)
```

Return the generator of the number field with parent object R.

`degree(R::AnticNumberField)`

Return the degree of the number field.

`signature(R::AnticNumberField)`

Returns the signature of the number field, i.e. a tuple (r, s) where r is the number of complex embeddings of R and s is half the number of complex embeddings of R.

`iszero(d::nf_elem)`

Return `true` if the given element d is equal to 0, else return `false`.

`isone(d::nf_elem)`

Return `true` if the given element d is equal to 1, else return `false`.

`isgen(d::nf_elem)`

Return `true` if the given element d is the generator of the number field it belongs to, else return `false`.

`coeff(d::nf_elem, n::int)`

Return the coefficient of degree n of the polynomial representation of the number field element d. If $n < 0$ we throw a `DomainError()`. If n is bigger than the degree of the element d we return 0.

`deepcopy(d::nf_elem)`

Return a new number field element arithmetically equal to d.

Examples.

Here are some examples of basic manipulation.

```
R, x = PolynomialRing(QQ, "x")
K, a = NumberField(x^3 + 3x + 1, "a")

d = a^2 + 2a - 7

k = zero(K)
l = one(K)
m = gen(K)
n = deepcopy(d)
p = coeff(d, 1)
isgen(m) == true
isone(l) == true
q = degree(K)
(r, s) = signature(K)
```

10.1.4 Unary operators

`-(d::nf_elem)`
Return $-d$.

Examples.

Here is an example of unary operators.

```
R, x = PolynomialRing(QQ, "x")
K, a = NumberField(x^3 + 3x + 1, "a")

d = a^2 + 2a - 7

f = -d
```

10.1.5 Binary operators

`+(c::nf_elem, d::nf_elem)`
Return $c + d$.

`-(c::nf_elem, d::nf_elem)`

Return $c - d$.

`*(c::nf_elem, d::nf_elem)`

Return cd.

Examples.

Here are some examples of binary operators.

```
R, x = PolynomialRing(QQ, "x")
K, a = NumberField(x^3 + 3x + 1, "a")

c = a^2 + 2a - 7
d = 3a^2 - a + 1

f = c + d
g = c - d
h = c*d
```

10.1.6 Ad hoc binary operators

```
+(c::nf_elem, d::Int)
+(c::nf_elem, d::fmpz)
+(c::nf_elem, d::fmpq)
```

Return $c + d$ where d is coerced into the number field.

```
+(c::Int, d::nf_elem)
+(c::fmpz, d::nf_elem)
+(c::fmpq, d::nf_elem)
```

Return $c + d$ where c is coerced into the number field.

```
-(c::nf_elem, d::Int)
-(c::nf_elem, d::fmpz)
-(c::nf_elem, d::fmpq)
```

Return $c - d$ where d is coerced into the number field.

```
-(c::Int, d::nf_elem)
-(c::fmpz, d::nf_elem)
-(c::fmpq, d::nf_elem)
```

Return $c - d$ where c is coerced into the number field.

```
*(c::nf_elem, d::Int)
*(c::nf_elem, d::fmpz)
*(c::nf_elem, d::fmpq)
```

Return cd where d is coerced into the number field.

```
*(c::Int, d::nf_elem)
*(c::fmpz, d::nf_elem)
*(c::fmpq, d::nf_elem)
```

Return cd where c is coerced into the number field.

Examples.

Here is an example of ad hoc binary operators.

```
R, x = PolynomialRing(QQ, "x")
K, a = NumberField(x^3 + 3x + 1, "a")

d = a^2 + 2a - 7

f = d + 21
g = fmpz(3) + d
h = d - fmpq(2, 3)
k = fmpq(2, 7)*d
l = d*fmpz(5)
```

10.1.7 Powering

`^(d::nf_elem, n::Int)`
Return d^n.

Examples.

Here are some examples of powering.

```
R, x = PolynomialRing(QQ, "x")
K, a = NumberField(x^3 + 3x + 1, "a")

d = a^2 + 2a - 7

f = d^5
g = d^(-2)
h = d^0
```

10.1.8 Comparison

`==(c::nf_elem, d::nf_elem)`
Return `true` if c is equal to d in the number field.

Examples.

Here are some examples of comparison.

```
R, x = PolynomialRing(QQ, "x")
K, a = NumberField(x^3 + 3x + 1, "a")

c = 3a^2 - a + 1
d = a^2 + 2a - 7

c != d
c == 3a^2 - a + 1
```

10.1.9 Inversion

`inv(c::nf_elem)`
Return the multiplicative inverse of c in the number field. If $c = 0$ we throw a `DivideError()`.

Examples.

Here is an example of inversion.

```
R, x = PolynomialRing(QQ, "x")
K, a = NumberField(x^3 + 3x + 1, "a")

c = 3a^2 - a + 1

c*inv(c) == 1
```

10.1.10 Exact division

```
divexact(c::nf_elem, d::nf_elem)
```

Return c times the inverse of d in the number field. If $d = 0$ we throw a `DivideError()`.

Examples.

Here are some examples of exact division.

```
R, x = PolynomialRing(QQ, "x")
K, a = NumberField(x^3 + 3x + 1, "a")

c = 3a^2 - a + 1
d = a^2 + 2a - 7

divexact(c, d) == c*inv(d)
```

10.1.11 Ad hoc exact division

```
divexact(c::nf_elem, d::Int)
divexact(c::nf_elem, d::fmpz)
divexact(c::nf_elem, d::fmpq)
```

Return c divided by d in the number field. If $d = 0$ we throw a `DivideError()`.

Examples.

Here are some examples of ad hoc exact division.

```
R, x = PolynomialRing(QQ, "x")
K, a = NumberField(x^3 + 3x + 1, "a")

c = 3a^2 - a + 1

divexact(7c, 7) == c
divexact(fmpq(2, 3)*c, fmpq(2, 3)) == c
```

10.1.12 Norm and trace

```
norm(c::nf_elem)
```

Return the norm of the given number field element c as an `fmpq`.

```
trace(c::nf_elem)
```

Return the trace of the given number field element c as an `fmpq`.

Examples.

Here are some examples of norm and trace.

```
R, x = PolynomialRing(QQ, "x")
K, a = NumberField(x^3 + 3x + 1, "a")

c = 3a^2 - a + 1

norm(c)
trace(c)
```

11 Arb rings

Arb is a library for real and complex numbers implemented using a ball (mid-rad) representation which tracks error bounds rigorously.

11.1 Arb real numbers ($\mathbb{R}$): arb

Arb provides the module `arb` for real numbers represented in mid-rad interval form $[m \pm r] = [m-r, m+r]$. The type of Arb real numbers in Nemo is also given the name `arb`.

The parent object for Arb reals is of type `ArbField`, which belongs to the `Field` type class.

The `ArbField` parent is initialised with the precision in bits (`prec`) used for operations on interval midpoints. The precision used for interval radii is a fixed implementation-defined constant (30 bits).

The following creates a parent object `RR` representing the field of real numbers with 64-bit precision, and elements `x`, `y` representing the rational numbers 1/4 and 1/10:

```
RR = ArbField(64)
x = RR("0.25")
y = RR("0.1")
```

Since interval midpoints are binary floating-point numbers with finite precision, one has `isexact(x) == true` and `isexact(y) == false` in this example.

In the following, we will assume that `RR` is an `ArbField` instance as above.

11.1.1 Constructors

```
RR(x::Float64)
RR(x::Int)
RR(x::UInt)
RR(x::fmpz)
RR(x::fmpq)
RR(x::arb)
```

Constructs an `arb` element from the given value.

```
ball(m::arb, r::arb)
```

Constructs an `arb` enclosing the range $[m - |r|, m + |r|]$ given the pair (m, r).

```
RR(x::String)
```

Constructs an `arb` element from a string representation. Valid strings include integer and floating-point literals. Midpoint-radius pairs of such literals may also be given in the form `"m +/- r"` or `"[m +/- r]"`

Examples.

```
x = RR(3)
y = RR(QQ(2,3))

z = ball(RR(3), RR("0.0001"))

s = RR("3 +/- 0.0001")
t = RR("-1.24e+12345")

u = RR("nan +/- inf")
```

11.1.2 Conversions

`Float64(x::arb)`
Return the midpoint of x rounded down to a machine double.

Examples.

```
x = RR(QQ(2,3))

Float64(x)
```

11.1.3 Parts of numbers

`midpoint(x::arb)`

`radius(x::arb)`
Return the midpoint and the radius of the interval x, respectively. The output is a point-valued arb instance, having radius zero.

Examples.

```
x = RR(QQ(2,3))

midpoint(x)

radius(x)
```

11.1.4 Comparisons and predicates

Boolean-valued functions defined on intervals must be used carefully.

In general, we extend predicates defined on point values (such as the usual comparisons $=, \neq, \leq, \ldots$ defined on pairs of real numbers) to intervals as follows: we return `true` if the predicate holds for all choices of points in the input intervals, and therefore certainly is true. We return `false` otherwise, i.e. both if the predicate either certainly is false or if truth cannot be determined.

For example, we certainly have $[0 \pm 0.1] \neq 1$, but both the comparisons $[0 \pm 0.1] \neq 0$ and $[0 \pm 0.1] = 0$ return `false`.

```
iszero(x::arb)
isnonzero(x::arb)
isone(x::arb)
isfinite(x::arb)
isexact(x::arb)
isint(x::arb)
ispositive(x::arb)
isnonnegative(x::arb)
isnegative(x::arb)
isnonpositive(x::arb)
```

Return whether x certainly is, respectively, equal to zero, not equal to zero, equal to 1, finite (having finite midpoint and radius), exact (having zero radius), an exact integer, positive, nonnegative, negative, or nonpositive.

```
overlaps(x::arb, y::arb)
```

Return whether x and y have any point in common.

```
contains(x::arb, y::fmpq)
contains(x::arb, y::fmpz)
contains(x::arb, y::Int)
contains(x::arb, y::BigFloat)
contains(x::arb, y::arb)
```

Return whether the interval x contains the point y, or entirely contains y when this is an interval.

```
contains_zero(x::arb)
contains_positive(x::arb)
contains_negative(x::arb)
contains_nonpositive(x::arb)
contains_nonnegative(x::arb)
```

Return whether the interval x respectively contains the point x, any positive point, any negative point, any nonnegative point, or any nonnegative point.

```
==(x::arb, y::arb)
!=(x::arb, y::arb)
>(x::arb, y::arb)
>=(x::arb, y::arb)
<(x::arb, y::arb)
<=(x::arb, y::arb)
```

Compare two intervals.

```
==(x::arb, y::Int)
!=(x::arb, y::Int)
<=(x::arb, y::Int)
>=(x::arb, y::Int)
<(x::arb, y::Int)
>(x::arb, y::Int)

==(x::Int, y::arb)
!=(x::Int, y::arb)
<=(x::Int, y::arb)
>=(x::Int, y::arb)
<(x::Int, y::arb)
>(x::Int, y::arb)

==(x::arb, y::fmpz)
!=(x::arb, y::fmpz)
<=(x::arb, y::fmpz)
>=(x::arb, y::fmpz)
<(x::arb, y::fmpz)
>(x::arb, y::fmpz)

==(x::fmpz, y::arb)
!=(x::fmpz, y::arb)
<=(x::fmpz, y::arb)
>=(x::fmpz, y::arb)
<(x::fmpz, y::arb)
>(x::fmpz, y::arb)

==(x::arb, y::Float64)
!=(x::arb, y::Float64)
<=(x::arb, y::Float64)
>=(x::arb, y::Float64)
<(x::arb, y::Float64)
>(x::arb, y::Float64)

==(x::Float64, y::arb)
!=(x::Float64, y::arb)
<=(x::Float64, y::arb)
>=(x::Float64, y::arb)
<(x::Float64, y::arb)
>(x::Float64, y::arb)
```

Compare an interval with a quantity of another type.

```
strongequal(x::arb, y::arb)
```

Returns whether x and y are identical as intervals, that is, have identical components midpoints and radii. Note that this is not the same as testing mathematical equality of real numbers, as implemented by the == operator.

Examples.

```
x = RR("1 +/- 0.001")
y = RR("3")
z = RR("4")

!isexact(x)

isint(y)

x <= z

x == 3
```

11.1.5 Arithmetic operations

```
-(x::arb)
abs(x::arb)
inv(x::arb)

+(x::arb, y::arb)
-(x::arb, y::arb)
*(x::arb, y::arb)
//(x::arb, y::arb)

+(x::arb, y::UInt)
-(x::arb, y::UInt)
*(x::arb, y::UInt)
//(x::arb, y::UInt)

+(x::UInt, y::arb)
-(x::UInt, y::arb)
*(x::UInt, y::arb)
//(x::UInt, y::arb)

+(x::arb, y::Int)
-(x::arb, y::Int)
*(x::arb, y::Int)
//(x::arb, y::Int)

+(x::Int, y::arb)
```

```
-(x::Int, y::arb)
*(x::Int, y::arb)
//(x::Int, y::arb)

+(x::arb, y::fmpz)
-(x::arb, y::fmpz)
*(x::arb, y::fmpz)
//(x::arb, y::fmpz)

+(x::fmpz, y::arb)
-(x::fmpz, y::arb)
*(x::fmpz, y::arb)
//(x::fmpz, y::arb)

^(x::arb, y::arb)
^(x::arb, y::fmpz)
^(x::arb, y::Int)
^(x::arb, y::UInt)
^(x::arb, y::fmpq)
```

Perform the respective arithmetic involving real numbers.

Examples.

```
x = RR(3)
y = RR(QQ(2,3))

x + y

inv(y) - x

3 * x + ZZ(100) // y

(x^2 + y^2) ^ QQ(1,2)
```

11.1.6 Miscellaneous operations

```
ldexp(x::arb, y::Int)

ldexp(x::arb, y::fmpz)
```

Return x multiplyed by 2^y exactly.

```
trim(x::arb)
```

Return an `arb` interval containing x but which may be more economical, by rounding off insignificant bits from the midpoint.

```
accuracy_bits(x::arb)
```

Return the relative accuracy of x measured in bits, capped between `typemax(Int)` and `-typemax(Int)`.

`unique_integer(x::arb)`

Return a pair where the first value is a boolean and the second is a `fmpz` integer. The boolean indicates whether the interval x contains a unique integer. If this is the case, the second return value is set to this unique integer.

`setunion(x::arb, y::arb)`

Return an `arb` containing the union of the intervals represented by `x` and `y`.

Examples.

```
x = RR("1 +/- 0.0001")
y = RR("2")

ldexp(x, 100)

accuracy_bits(x)
accuracy_bits(y)

unique_integer(y)

setunion(x, y)
```

11.1.7 Mathematical constants

`const_pi(r::ArbField)`

Return $\pi = 3.14159\ldots$ as an element of r.

`const_e(r::ArbField)`

Return $e = 2.71828\ldots$ as an element of r.

`const_log2(r::ArbField)`

Return $\log(2) = 0.69314\ldots$ as an element of r.

`const_log10(r::ArbField)`

Return $\log(10) = 2.302585\ldots$ as an element of r.

`const_euler(r::ArbField)`

Return Euler's constant $\gamma = 0.577215\ldots$ as an element of r.

`const_catalan(r::ArbField)`

Return Catalan's constant $C = 0.915965\ldots$ as an element of r.

```
const_khinchin(r::ArbField)
```

Return Khinchin's constant $K = 2.685452\ldots$ as an element of r.

```
const_glaisher(r::ArbField)
```

Return Glaisher's constant $A = 1.282427\ldots$ as an element of r.

Examples.

```
x = const_pi(RR)

y = const_pi(ArbField(3323))

g = const_euler(R)
```

11.1.8 Mathematical functions

```
floor(x::arb)
ceil(x::arb)
```

Return the floor and ceiling functions $\lfloor x \rfloor$ and $\lceil x \rceil$ respectively.

```
sqrt(x::arb)
```

Return $\sqrt{x}$.

```
rsqrt(x::arb)
```

Return $1/\sqrt{x}$.

```
sqrt1pm1(x::arb)
```

Return $\sqrt{1+x} - 1$, evaluated accurately for small x.

```
hypot(x::arb, y::arb)
```

Return $\sqrt{x^2 + y^2}$.

```
root(x::arb, n::UInt)
root(x::arb, n::Int)
```

Return the principal branch of $x^{1/n}$. Require that $n > 0$.

```
log(x::arb)
```

Return the principal branch of $\log(x)$.

```
log1p(x::arb)
```

Return $\log(1+x)$, evaluated accurately for small x.

```
exp(x::arb)
```

Return $\exp(x)$.

```
expm1(x::arb)
```

Return $\exp(x) - 1$, evaluated accurately for small x.

```
sin(x::arb)
cos(x::arb)
tan(x::arb)
cot(x::arb)
```

Return the respective trigonometric function $\sin(x)$, etc.

```
sinpi(x::arb)
cospi(x::arb)
tanpi(x::arb)
cotpi(x::arb)
```

Return the respective trigonometric function $\sin(\pi x)$, etc.

```
sincos(x::arb)
sincospi(x::arb)
```

Return a pair containing both the sine and cosine computed simultaneously.

```
sinpi(x::fmpq, r::ArbField)
cospi(x::fmpq, r::ArbField)
sincospi(x::fmpq, r::ArbField)
```

Return the sine and/or cosine of argument πx as an element of r.

```
sinh(x::arb)
cosh(x::arb)
tanh(x::arb)
coth(x::arb)
sinhcosh(x::arb)
```

Return the respective hyperbolic function $\sinh(x)$, etc.

```
atan(x::arb)
asin(x::arb)
acos(x::arb)
atanh(x::arb)
asinh(x::arb)
acosh(x::arb)
```

Return the respective inverse trigonometric or hyperbolic function $\operatorname{atan}(x)$, etc.

```
atan2(b::arb, a::arb)
```

Return $\operatorname{atan2}(b, a) = \arg(a + bi)$.

```
gamma(x::arb)
```

Return the gamma function $\Gamma(x)$.

```
gamma(x::fmpz, r::ArbField)
gamma(x::fmpq, r::ArbField)
```

Return the gamma function of x as an element of r.

```
lgamma(x::arb)
```

Return the logarithm of the gamma function, $\log \Gamma(x)$.

```
rgamma(x::arb)
```

Return the reciprocal of the gamma function, $1/\Gamma(x)$.

```
digamma(x::arb)
```

Return the logarithmic derivative of the gamma function, $\psi(x)$.

```
fac(x::arb)
```

Return the factorial $x! = \Gamma(x + 1)$.

```
fac(n::UInt, r::ArbField)
fac(n::Int, r::ArbField)
```

Return the factorial $n!$ as an element of r.

```
risingfac(x::arb, n::UInt)
risingfac(x::arb, n::Int)
```

Return the rising factorial $x(x + 1) \cdots (x + n - 1)$.

```
risingfac2(x::arb, n::UInt)
risingfac2(x::arb, n::Int)
```

Return a tuple containing the rising factorial and its derivative.

```
risingfac(x::fmpq, n::UInt, r::ArbField)
risingfac(x::fmpq, n::Int, r::ArbField)
```

Return the rising factorial $x(x+1)\cdots(x+n-1)$ as an element of r.

```
binom(x::arb, k::UInt)
```

Return the binomial coefficient $\binom{x}{k}$.

```
binom(n::UInt, k::UInt, r::ArbField)
```

Return the binomial coefficient $\binom{n}{k}$ as an element of r.

```
fib(n::fmpz, r::ArbField)
fib(n::UInt, r::ArbField)
fib(n::Int, r::ArbField)
```

Return the Fibonacci number F_n as an element of r.

```
bell(n::fmpz, r::ArbField)
bell(n::Int, r::ArbField)
```

Return the Bell number B_n as an element of r.

```
bernoulli(n::UInt, r::ArbField)
bernoulli(n::Int, r::ArbField)
```

Return the Bernoulli number B_n as an element of r.

```
chebyshev_t(n::UInt, x::arb)
chebyshev_u(n::UInt, x::arb)
chebyshev_t(n::Int, x::arb)
chebyshev_u(n::Int, x::arb)
```

Return the value of the Chebyshev polynomial $T_n(x)$ or $U_n(x)$ respectively.

```
chebyshev_t2(n::UInt, x::arb)
chebyshev_u2(n::UInt, x::arb)
chebyshev_t2(n::Int, x::arb)
chebyshev_u2(n::Int, x::arb)
```

Return the tuple $(T_n(x), T_{n-1}(x))$ or $(U_n(x), U_{n-1}(x))$, both values computed simultaneously.

```
zeta(s::arb)
```

Return the Riemann zeta function $\zeta(s)$.

```
zeta(n::UInt, r::ArbField)
zeta(n::Int, r::ArbField)
```

Return the Riemann zeta function $\zeta(n)$ as an element of r.

```
zeta(s::arb, a::arb)
```

Return the Hurwitz zeta function $\zeta(s, a)$.

```
polylog(s::arb, z::arb)
polylog(s::Int, z::arb)
```

Return the polylogarithm $\mathrm{Li}_s(z)$.

```
agm(x::arb, y::arb)
```

Return the arithmetic-geometric mean of x and y.

Examples.

```
x = floor(exp(RR(1)))

x = sinpi(QQ(5,6), RR)

y = gamma(QQ(1,3), ArbField(256))

z = bernoulli(1000, ArbField(53))

w = polylog(3, RR(-10))
```

11.2 Arb complex numbers ($\mathbb{C}$): acb

Arb provides the module acb for complex numbers represented in rectangular form $a + bi$ where a, b are arb numbers. The type of Arb complex numbers in Nemo is also given the name acb.

The parent object for Arb complex numbers is of type AcbField, which belongs to the Field type class.

The AcbField parent is initialised with the precision in bits (prec) used for operations on interval midpoints.

The following creates a parent object CC representing the field of complex numbers with 64-bit precision, and an element $x = 2 + 3i$:

```
CC = AcbField(64)
x = CC(2,3)
```

In the following, we will assume that CC is an AcbField instance as above, and that RR is an ArbField instance.

11.2.1 Constructors

```
CC(x::Int)
CC(x::UInt)
CC(x::fmpz)
CC(x::fmpq)
CC(x::arb)
CC(x::Float64)
CC(x::arb, y::arb)
CC(x::Int, y::Int)
CC(x::AbstractString)
CC(x::AbstractString, y::AbstractString)
```

Constructs an `acb` element from the given real number or pair of real numbers.

```
zero(r::AcbField)
one(r::AcbField)
onei(r::AcbField)
```

Respectively return 0, 1, $i = \sqrt{-1}$ as an element of r.

Examples.

```
x = CC(3)

y = CC(QQ(2,3))

z = CC(RR(3), RR(4))

w = CC("1.23", "5.6")
```

11.2.2 Complex parts

```
real(x::acb)
imag(x::acb)
abs(x::acb)
angle(x::acb)
```

Real part, imaginary part, absolute value, phase. These functions return a real number of type `arb`, belonging to an `ArbField` with the same precision as the parent of x.

```
conj(x::acb)
```

Complex conjugate of x.

11.2.3 Comparisons and predicates

See the section for `arb` numbers for definitions of boolean-valued functions for intervals.

```
iszero(x::acb)
isone(x::acb)
isfinite(x::acb)
isexact(x::acb)
isint(x::acb)
isreal(x::acb)
```

Return whether x certainly is, respectively, equal to zero, equal to 1, finite, exact, an exact integer, purely real (having zero imaginary part).

```
overlaps(x::acb, y::acb)
```

Return whether x and y have any point in common.

```
contains(x::acb, y::fmpq)
contains(x::acb, y::fmpz)
contains(x::acb, y::Int)
contains(x::acb, y::acb)
contains_zero(x::acb)
```

Return whether the complex interval x contains the point y, or entirely contains y when this is a complex interval.

```
==(x::acb,y::acb)
!=(x::acb,y::acb)

==(x::acb,y::Int)
==(x::Int,y::acb)
==(x::acb,y::arb)
==(x::arb,y::acb)
==(x::acb,y::fmpz)
==(x::fmpz,y::acb)
==(x::acb,y::arb)
==(x::arb,y::acb)
==(x::acb,y::Float64)
==(x::Float64,y::acb)
!=(x::acb,y::Int)
!=(x::Int,y::acb)
!=(x::acb,y::arb)
!=(x::arb,y::acb)
!=(x::acb,y::fmpz)
!=(x::fmpz,y::acb)
!=(x::acb,y::arb)
!=(x::arb,y::acb)
!=(x::acb,y::Float64)
!=(x::Float64,y::acb)
```

Compare for equality or inequality.

```
strongequal(x::acb, y::acb)
```

Returns whether x and y are identical as intervals, that is, have identical midpoints and radii for the real and imaginary parts. Note that this is not the same as testing mathematical equality of complex numbers, as implemented by the == operator.

Examples.

```
x = CC("1 +/- 0.001")
y = CC("3")
z = CC(4,5)

!isexact(x)
isexact(z)

isint(y)
isreal(x)

contains(x, 1)
!overlaps(y,z)

x == 1
x != y
```

11.2.4 Arithmetic operations

```
-(x::acb)
inv(x::acb)
```

Additive and multiplicative inverse.

```
+(x::acb, y::acb)
-(x::acb, y::acb)
*(x::acb, y::acb)
//(x::acb, y::acb)
^(x::acb, y::acb)

+(x::acb, y::Int)
-(x::acb, y::Int)
*(x::acb, y::Int)
//(x::acb, y::Int)
^(x::acb, y::Int)

+(x::Int, y::acb)
-(x::Int, y::acb)
*(x::Int, y::acb)
//(x::Int, y::acb)
^(x::Int, y::acb)

+(x::acb, y::UInt)
-(x::acb, y::UInt)
*(x::acb, y::UInt)
//(x::acb, y::UInt)
^(x::acb, y::UInt)

+(x::UInt, y::acb)
-(x::UInt, y::acb)
```

```
*(x::UInt, y::acb)
//(x::UInt, y::acb)
^(x::UInt, y::acb)

+(x::acb, y::fmpz)
-(x::acb, y::fmpz)
*(x::acb, y::fmpz)
//(x::acb, y::fmpz)
^(x::acb, y::fmpz)

+(x::fmpz, y::acb)
-(x::fmpz, y::acb)
*(x::fmpz, y::acb)
//(x::fmpz, y::acb)
^(x::fmpz, y::acb)

+(x::acb, y::fmpq)
-(x::acb, y::fmpq)
*(x::acb, y::fmpq)
//(x::acb, y::fmpq)
^(x::acb, y::fmpq)

+(x::fmpq, y::acb)
-(x::fmpq, y::acb)
*(x::fmpq, y::acb)
//(x::fmpq, y::acb)
^(x::fmpq, y::acb)

+(x::acb, y::arb)
-(x::acb, y::arb)
*(x::acb, y::arb)
//(x::acb, y::arb)
^(x::acb, y::arb)

+(x::arb, y::acb)
-(x::arb, y::acb)
*(x::arb, y::acb)
//(x::arb, y::acb)
^(x::arb, y::acb)
```

Arithmetic operations with `acb` numbers and simpler types, producing `acb` numbers as results. Complex powers evaluate to the principal branch.

Examples.

```
z = CC(10,3)
w = CC(3,4)

z^2 + w ^ QQ(1,2) + z^w
```

```
-z * inv(z) + 1

3*z + 4*w + 5*real(z)
```

11.2.5 Miscellaneous operations

```
ldexp(x::acb, y::Int)

ldexp(x::acb, y::fmpz)
```

Return x multiplyed by 2^y exactly.

```
trim(x::acb)
```

Return an `acb` interval containing x but which may be more economical, by rounding off insignificant bits from the midpoints.

```
accuracy_bits(x::acb)
```

Return the relative accuracy of x measured in bits, capped between `typemax(Int)` and `-typemax(Int)`.

```
unique_integer(x::acb)
```

Return a pair where the first value is a boolean and the second is a `fmpz` integer. The boolean indicates whether the interval x contains a unique integer. If this is the case, the second return value is set to this unique integer.

Examples.

```
x = CC("1 +/- 0.0001", "0 +/- 1e-30")
y = CC("2")

ldexp(x, 100)

accuracy_bits(x)
accuracy_bits(y)

unique_integer(y)
```

11.2.6 Mathematical constants

```
const_pi(r::AcbField)
```

Return $\pi = 3.14159\ldots$ as an element of r.

11.2.7 Mathematical functions

`sqrt(x::acb)`
Return $\sqrt{x}$.

`rsqrt(x::acb)`

Return $1/\sqrt{x}$.

`log(x::acb)`

Return the principal branch of $\log(x)$.

`log1p(x::acb)`

Return $\log(1+x)$, evaluated accurately for small x.

`exp(x::acb)`

Return $\exp(x)$.

`exppii(x::acb)`

Return $\exp(\pi i x)$.

`sin(x::acb)`
`cos(x::acb)`
`tan(x::acb)`
`cot(x::acb)`

Return the respective trigonometric function $\sin(x)$, etc.

`sinpi(x::acb)`
`cospi(x::acb)`
`tanpi(x::acb)`
`cotpi(x::acb)`

Return the respective trigonometric function $\sin(\pi x)$, etc.

`sincos(x::acb)`
`sincospi(x::acb)`

Return a pair containing both the sine and cosine computed simultaneously.

`sinh(x::acb)`
`cosh(x::acb)`
`tanh(x::acb)`
`coth(x::acb)`
`sinhcosh(x::acb)`

Return the respective hyperbolic function $\sinh(x)$, etc.

```
atan(x::acb)
```

Return the inverse tangent $\operatorname{atan}(x)$.

```
logsinpi(x::acb)
```

Return $\log \sin(\pi x)$, constructed without branch cuts off the real line.

```
gamma(x::acb)
```

Return the gamma function $\Gamma(x)$.

```
gamma(x::fmpz, r::ArbField)
gamma(x::fmpq, r::ArbField)
```

Return the gamma function of x as an element of r.

```
lgamma(x::acb)
```

Return the logarithm of the gamma function, $\log \Gamma(x)$.

```
rgamma(x::acb)
```

Return the reciprocal of the gamma function, $1/\Gamma(x)$.

```
digamma(x::acb)
```

Return the logarithmic derivative of the gamma function, $\psi(x)$.

```
risingfac(x::acb, n::UInt)
risingfac(x::acb, n::Int)
```

Return the rising factorial $x(x+1)\cdots(x+n-1)$.

```
risingfac2(x::acb, n::UInt)
risingfac2(x::acb, n::Int)
```

Return a tuple containing the rising factorial and its derivative.

```
polygamma(s::acb, z::acb)
```

Return the generalised polygamma function $\psi(s, z)$.

```
zeta(s::acb)
```

Return the Riemann zeta function $\zeta(s)$.

```
zeta(s::acb, a::acb)
```

Return the Hurwitz zeta function $\zeta(s, a)$.

```
polylog(s::acb, z::acb)
polylog(s::Int, z::acb)
```

Return the polylogarithm $\mathrm{Li}_s(z)$.

```
barnesg(s::acb)
```

Return the Barnes G-function $G(s)$.

```
logbarnesg(s::acb)
```

Return the logarithm of the Barnes G-function.

```
erf(x::acb)
erfi(x::acb)
erfc(x::acb)
```

Return the error function, imaginary error function, complementary error function.

```
ei(x::acb)
si(x::acb)
ci(x::acb)
shi(x::acb)
chi(x::acb)
li(x::acb)
lioffset(x::acb)
```

Return the exponential integral, sine integral, cosine integral hyperbolic sine integral, hyperbolic cosine integral, logarithmic integral, offset logarithmic integral.

```
expint(s::acb, x::acb)
```

Return the generalised exponential integral $E_s(x)$.

```
gamma(s::acb, x::acb)
```

Return the upper incomplete gamma function $\Gamma(s, x)$.

```
besselj(nu::acb, x::acb)
bessely(nu::acb, x::acb)
besseli(nu::acb, x::acb)
besselk(nu::acb, x::acb)
```

Return the Bessel function $J_\nu(x), Y_\nu(x), I_\nu(x), K_\nu(x)$.

```
hyp1f1(a::acb, b::acb, x::acb)
```

Return the confluent hypergeometric function ${}_1F1(a, b, x)$.

```
hyp1f1r(a::acb, b::acb, x::acb)
```

Return the regularized confluent hypergeometric function ${}_1F1(a, b, x)/\Gamma(b)$.

```
hyperu(a::acb, b::acb, x::acb)
```

Return the confluent hypergeometric function $U(a, b, x)$.

```
jtheta(z::acb, tau::acb)
```

Return a tuple of four elements containing the Jacobi theta function values $\theta_1, \theta_2, \theta_3, \theta_4$ evaluated at z, τ.

```
modeta(tau::acb)
modj(tau::acb)
modlambda(tau::acb)
moddelta(tau::acb)
```

Return the Dedekind eta function $\eta(\tau)$, the j-invariant $j(\tau)$, the modular lambda function $\lambda(\tau)$, the modular delta function $\Delta(\tau)$.

```
ellipwp(z::acb, tau::acb)
```

Return the Weierstrass elliptic function $\wp(z, \tau)$.

```
ellipk(z::acb)
ellipe(z::acb)
```

Return the complete elliptic integral $K(z)$, $E(z)$.

```
agm(x::acb)
agm(x::acb, y::acb)
```

Return the arithmetic-geometric mean of x and y. With one argument, return the AGM of 1 and x.

Examples.

```
s = CC(1, 2)
z = CC("1.23", "3.45")

sin(z)^2 + cos(z)^2

zeta(z)

besselj(s, z)

hyp1f1(s, s+1, z)
```

12 Pari rings

Pari is a library for algebraic number theory, maintained in Bordeaux, France. We currently use Pari to provide arithmetic in maximal orders (rings of integers) of number fields and ideals of number fields.

Pari is dynamically typed, though on a lower level it has specific types for a range of rings and fields. Many Pari objects are however represented not by special objects, but simply by their low level representation in terms of vectors of lower level objects, etc.

We give Nemo types to Pari objects where they have none in Pari.

12.1 Pari maximal orders: `pari_maximal_order_elem`

Pari has no specific type for an element of a maximal order. In Nemo a maximal order parent object has type `PariMaximalOrder`, which belongs to the `Ring` type class. Elements of a maximal order are given the type `pari_maximal_order_elem` which belongs to the `MaximalOrderElem` type class, which in turn belongs to `RingElem`.

The user need not deal with the types directly since we provide a function `PariMaximalOrder` to create parent objects for Pari maximal orders and we also provide various constructors to create elements of such orders.

By default, the global variable `MaximalOrder` is set equal to `PariMaximalOrder` which means that the following code can be used to create maximal order parent objects in Nemo.

```
R, x = PolynomialRing(QQ, "x")
K, a = NumberField(x^3 + 3x + 1, "a")
O = MaximalOrder(K)
```

Here, $x^3 + 3x + 1$ is the irreducible polynomial over $\mathbb{Q}$ giving the minimum polynomial of the generator a of the number field K (though it need not be a monic polynomial).

Note that at the present time, Pari maximal orders are constructed from Antic number fields. This may change in a later version of Nemo.

12.1.1 Constructors

Various constructors are provided to create elements of a Pari maximal order. In the constructors below we assume that O is the parent object of such a maximal order, for example created by the construction above.

`O()`

Create the element 0 of the number field.

`O(d::Integer)`
`O(d::fmpz)`

Create the element of the maximal order which results from mapping the integer d into the maximal order via the embedding of the integers into the order.

`O(d::fmpq_poly)`

Given a polynomial d, consider the polynomial to be an element of the number field K and embed it into the maximal order O. The element is represented in terms of a basis for O.

```
O(d::nf_elem)
```

Embed the number field element d into the maximal order O. The element is represented in terms of a basis for O.

```
O(d::pari_maximal_order_elem)
```

Return a reference to the maximal order element d. No copy of the data is made.

Examples.

Here are some examples of constructors.

```
R, x = PolynomialRing(QQ, "x")
K, a = NumberField(x^3 + 3x + 1, "a")
O = MaximalOrder(K)

k = O()
l = O(123)
m = O(fmpz(12))
n = O(x^2 + 2x - 7)
p = O(a^2 + 2a - 7)
q = O(n)
```

12.1.2 Basic manipulation

```
basis(O::PariMaximalOrder)
```

Return the basis of the maximal order O. This is a vector of Pari polynomials representing the basis elements. They can be accessed using array notation and coerced into the ring of Flint polynomials over the rationals.

Examples.

Here are some examples of basic manipulation.

```
R, x = PolynomialRing(QQ, "x")
K, a = NumberField(x^3 + 3x + 1, "a")
O = MaximalOrder(K)

B = basis(O)
b3 = B[3]
c = R(b3)
```

13 Pari collections

Pari provides numerous objects which aren't rings or fields, but merely sets. We call such an object a collection in Nemo.

13.1 Pari ideals: PariIdeal

Ideals behave more like a parent object than an element object, thus we capitalise the name given to the types of both the parent object of an ideal and the ideal itself in Nemo.

Pari has no specific type for an ideal of a number field. In Nemo the parent object for an ideal collection has type PariIdealCollection, which belongs to the Collection type class. Note that even though ideals can be added and multiplied, they don't form a ring.

Elements of an ideal collection in Nemo are ideals, which are given the type PariIdeal which belongs directly to the CollectionElem type class.

The user need not deal with the types directly since we provide a function PariIdeal to create Pari ideal objects (not their parents) directly.

By default, the global variable Ideal is set equal to PariIdeal so that ideals can be constructed using Ideal.

13.1.1 Constructors

In the following constructors, we take O to be the parent object for a Pari maximal order, constructed for example by the following code.

```
R, x = PolynomialRing(QQ, "x")
K, a = NumberField(x^3 + 3x + 1, "a")
O = MaximalOrder(K)
```

We note that the constructors for Pari ideals objects below are variadic, meaning that they can take a comma separated list of generators for the ideal.

```
Ideal(O::PariMaximalOrder, b::Integer, ...)
```

Construct the ideal generated by the number field elements represented by the given variadic list of integers.

```
Ideal(O::PariMaximalOrder, b::fmpz, ...)
```

Construct the ideal generated by the number field elements represented by the given variadic list of integers.

```
Ideal(O::PariMaximalOrder, b::fmpq_poly, ...)
```

Construct the ideal generated by the number field elements represented by the given variadic list of polynomials.

```
Ideal(O::PariMaximalOrder, b::nf_elem, ...)
```

Construct the ideal generated by the given number field elements.

```
Ideal(O::PariMaximalOrder, b::pari_maximal_order_elem, ...)
```

Construct the ideal generated by the given maximal order elements.

Examples.

Here are some examples of constructors.

```
R, x = PolynomialRing(QQ, "x")
K, a = NumberField(x^3 + 3x + 1, "a")
O = MaximalOrder(K)

c = Ideal(O, 2, 3)
d = Ideal(O, x + 1, R(3))
f = Ideal(O, a + 1, K(3))
g = Ideal(O, O(a + 1), O(3))
```

13.1.2 Listing ideals

`bounded_ideals(R::PariMaximalOrder, bound::Int)`

Create a list of all ideals of the given order with norm at most equal to the given bound. The list is returned as a Julia array indexed by the norms from 1 to the given bound. Each entry in the array is a Pari vector of ideals which can be accessed using array notation.

`prime_decomposition(R::PariMaximalOrder, p::fmpz)`

Create a list of all ideals dividing the given prime number p. We do not check that p is prime, though it is required to be prime. The list is returned as a Pari vector of ideals which can be accessed using array notation.

Examples.

Here are some examples of ideal lists.

```
R, x = PolynomialRing(QQ, "x")
K, a = NumberField(x^3 + 3x + 1, "a")
O = MaximalOrder(K)

S = prime_decomposition(O, 13)
T = bounded_ideals(O, 100)

a = S[2]
b = T[97][1]
```

13.1.3 Basic manipulation

`numden(a::PariIdeal)`

Return a pair (n, d) consisting of the numerator and denominator of the given ideal (as ideals).

`valuation(a::PariIdeal, p::PariIdeal)`

Given a prime ideal p as returned by `prime_decomposition`, return the valuation of a at p.

Examples.

Here are some examples of basic manipulation.

```
R, x = PolynomialRing(QQ, "x")
K, a = NumberField(x^3 + 3x + 1, "a")
O = MaximalOrder(K)

S = Ideal(O, x + 1)
T = Ideal(O, fmpq(2, 3)*x + 1)
A = prime_decomposition(O, 3)

valuation(S, A[1])
(n, d) = numden(T)
```

13.1.4 Ideal norm

`norm(a::PariIdeal)`

Return the norm of the ideal as an `fmpz` integer.

Examples.

Here is an example of ideal norm.

```
R, x = PolynomialRing(QQ, "x")
K, a = NumberField(x^3 + 3x + 1, "a")
O = MaximalOrder(K)

S = Ideal(O, x + 1)

a = norm(S)
```

13.1.5 Binary operators

`+(a::PariIdeal, b::PariIdeal)`

Return the sum of the given ideals.

`*(a::PariIdeal, b::PariIdeal)`

Return the product of the given ideals.

`intersect(a::PariIdeal, b::PariIdeal)`

Return the intersection of the given ideals.

Examples.

Here are some examples of binary operators.

```
R, x = PolynomialRing(QQ, "x")
K, a = NumberField(x^3 + 3x + 1, "a")
O = MaximalOrder(K)

S = Ideal(O, x + 1)
T = Ideal(O, 3)

A = S + T
B = S*T
C = intersect(S, T)
```

13.1.6 Comparison

`==(a::PariIdeal, b::PariIdeal)`
Return `true` if the ideals are arithmetically the same ideal.

Examples.

Here are some examples of comparison.

```
R, x = PolynomialRing(QQ, "x")
K, a = NumberField(x^3 + 3x + 1, "a")
O = MaximalOrder(K)

S = Ideal(O, x + 1)
T = Ideal(O, 3)

S == S
S != T
```

13.1.7 Powering

`^(a::PariIdeal, n::Int)`
Return a^n.

Examples.

Here is an example of powering.

```
R, x = PolynomialRing(QQ, "x")
K, a = NumberField(x^3 + 3x + 1, "a")
O = MaximalOrder(K)

S = Ideal(O, x + 1)

T = S^3
```

13.1.8 Exact division

`divexact(a::PariIdeal, b::PariIdeal)`
Return the exact quotient of the ideals a and b.

Examples.

Here is an example of exact division.

```
R, x = PolynomialRing(QQ, "x")
K, a = NumberField(x^3 + 3x + 1, "a")
O = MaximalOrder(K)

S = Ideal(O, x + 1)
T = Ideal(O, 3)

divexact(S*T, T)
```

13.1.9 Inverse

`inv(a::PariIdeal)`
Return the inverse of the given ideal.

Examples.

Here is an example of inversion.

```
R, x = PolynomialRing(QQ, "x")
K, a = NumberField(x^3 + 3x + 1, "a")
O = MaximalOrder(K)

S = Ideal(O, x + 1)

inv(S)
```

13.1.10 Extended GCD

`gcdx(a::PariIdeal, b::PariIdeal)`
Given two coprime integer ideals a and b return a pair of values (s, t), with $s \in a$ and $t \in b$ such that $s + t = 1$. The values s and t will be returned as `fmpq_poly`'s.

Examples.

Here is an example of extended GCD.

```
R, x = PolynomialRing(QQ, "x")
K, a = NumberField(x^3 + 3x + 1, "a")
O = MaximalOrder(K)

S = Ideal(O, x + 1)
T = Ideal(O, 2x)

(s, t) = gcdx(S, T)
```

13.1.11 Ideal factorisation

`factor(a::PariIdeal)`

Return the prime factorisation of the given ideal. This is given as a Julia array of tuples (P, n) consisting of a prime ideal P and its exponent in the factorisation of a.

`factor_mul(a::PariFactor{pari_maximal_order_elem})`

Given a factorisation of an ideal a into prime factors as returned by `factor`, this function retrieves the original ideal a.

Examples.

Here is an example of ideal factorisation.

```
R, x = PolynomialRing(QQ, "x")
K, a = NumberField(x^3 + 3x + 1, "a")
O = MaximalOrder(K)

S = Ideal(O, 3)

A = factor(S)
T = factor_mul(A)
```

13.1.12 Ideal approximation

`approx(a::PariIdeal)`

Return an element of the given ideal which approximates the ideal, in the sense that the valuation at all prime ideals dividing a is the same, and the valuation at all other prime ideals is non-negative.

Examples.

Here is an example of ideal factorisation.

```
R, x = PolynomialRing(QQ, "x")
K, a = NumberField(x^3 + 3x + 1, "a")
O = MaximalOrder(K)

S = Ideal(O, 3)

f = approx(S)
```

13.1.13 Coprime ideal construction

`coprime_multiplier(x::PariIdeal, y::PariIdeal)`

Return an element of the number field b such that bx is an integral ideal coprime to the ideal y.

Examples.

Here is an example of coprime ideal construction.

```
R, x = PolynomialRing(QQ, "x")
K, a = NumberField(x^3 + 3x + 1, "a")
O = MaximalOrder(K)

S = Ideal(O, 3)
T = Ideal(O, x + 1)

f = coprime_multiplier(S, T)
```