Dom — predefined domains

Table of contents

`Dom::AlgebraicExtension` – simple algebraic field extensions

For a given field F and a polynomial $f \in F[x]$, `Dom::AlgebraicExtension(F, f, x)` creates the residue class field $F[x]/\langle f \rangle$.

`Dom::AlgebraicExtension(F, f1=f2, x)` does the same for $f = f_1 - f_2$.

Domain:

- `Dom::AlgebraicExtension(F, f)`
- `Dom::AlgebraicExtension(F, f, x)`
- `Dom::AlgebraicExtension(F, f1=f2)`
- `Dom::AlgebraicExtension(F, f1=f2, x)`

Parameters:

`F`	—	the ground field: a domain of category `Cat::Field`
`f, f1, f2`	—	polynomials or polynomial expressions
`x`	—	identifier

Details:

- `Dom::AlgebraicExtension(F, f, x)` creates the field $F[x]/\langle f \rangle$ of residue classes of polynomials modulo f. This field can also be written as $F(x)/\langle f \rangle$, the field of residue classes of rational functions modulo f.
- The parameter `x` may be omitted if `f` is a univariate polynomial or a polynomial expression that contains exactly one indeterminate; it is then taken to be the indeterminate occurring in `f`.
- The field `F` must have normal representation.
- `f` must not be a constant polynomial.
- `f` must be irreducible; this is *not* checked.
- `f` may be a polynomial over a coefficient ring different from `F`, or multivariate; however, it must be possible to convert it to a univariate polynomial over `F`. See Example 2.

`Dom::AlgebraicExtension(F, f)(g)` creates the residue class of `g` modulo `f`.

Creating Elements:

- `Dom::AlgebraicExtension(F,f)(g)`
- `Dom::AlgebraicExtension(F, f)(rat)`

Parameters:

`g` — element of the residue class to be defined: polynomial over `F` in the variable `x`, or any object convertible to such.

`rat` — rational function that belongs to the residue class to be defined: expression whose numerator and denominator can be converted to polynomials over `F` in the variable `x`. The denominator must not be a multiple of `f`.

Details:

- If `rat` has numerator and denominator `p` and `q`, respectively, then `Dom::AlgebraicExtension` equals `Dom::AlgebraicExtension(F,f)(p)` divided by `Dom::AlgebraicExtension(F,`

Categories:

`Cat::Field`, `Cat::Algebra(F)`, `Cat::VectorSpace(F)`,

if `F::hasProp(Cat::DifferentialRing)` then
`Cat::DifferentialRing`

if `F::hasProp(Cat::PartialDifferentialRing)` then
`Cat::PartialDifferentialRing`

Related Domains: `Dom::GaloisField`

Entries:

`zero` the zero element of the field extension

`one` the unit element of the field extension

`groundField` the ground field of the extension

`minpoly` the minimal polynomial `f`

`deg` the degree of the extension, i.e., of `f`

`variable` the unknown of the minimal polynomial `f`

`characteristic` the characteristic, which always equals the characteristic of the ground field. This entry only exists if the characteristic of the ground field is known.

`degreeOverPrimeField` the dimension of the field when viewed as a vector space over the prime field. This entry only exists if the ground field is a prime field, or its degree over its prime field is known.

Mathematical Methods

Method `_plus`: sum of field elements

```
_plus(dom a, ...)
```

- This method returns the sum of its arguments.
- This method overloads the function `_plus` of the system kernel.

Method `_mult`: product of field elements

```
_mult(any a, ...)
```

- This method returns the product of its arguments. The arguments must be either field elements, or convertible to such, or integers.
- This method overloads the function `_mult` of the system kernel.

Method `_negate`: negate a field element

```
_negate(dom a)
```

- This method returns the negative of `a`.
- This method overloads the function `_negate` of the system kernel.

Method `_subtract`: difference of field elements

```
_subtract(dom a, dom b)
```

- This method returns the difference $a - b$.
- This method overloads the function `_subtract` of the system kernel.

Method `iszero`: tests whether a field element is zero.

```
iszero(dom a)
```

- This method returns `TRUE` if `a` is known to be zero, and `FALSE` otherwise. If equality to zero can be decided for elements of the ground field, the answer `FALSE` implies that `a` cannot be zero.
- This method overloads the function `iszero`.

Method `intmult`: multiply a field element by an integer

```
intmult(dom a, integer b)
```

- This method computes `a*b`.
- This method is more efficient than `"_mult"` in this special case.

Method `_invert`: inverse of a field element

```
_invert(dom a)
```

- This method returns the inverse of `a` with respect to multiplication. `a` must be nonzero.
- This method overloads the function `_invert`.

Method `gcd`: gcd of field elements

```
gcd(dom a, ...)
```

- If the arguments are not all zero, any nonzero field element is a gcd. This method tries to find a gcd such that dividing the arguments by it gives most "simple" objects.
- This method overloads the function `gcd`.

Method `conjNorm`: norm of an element

```
conjNorm(dom a)
```

- This method computes the norm of `a`, i.e., the product of all conjugates of `a` (images of `a` under Galois automorphisms).

Method `conjTrace`: trace of an element

```
conjTrace(dom a)
```

- This method computes the trace of `a`, i.e., the sum of all conjugates of `a` (images of `a` under Galois automorphisms).

Method `minimalPolynomial`: minimal polynomial of an element

```
minimalPolynomial(dom a)
```

- This method chooses a free identifier and computes the (uniquely determined) monic irreducible polynomial over the ground field in that variable of which `a` is a root.

Method `D`: differential operator

```
D(dom a)
```

- This method implements the continuation of the differential operator of the ground field; it exists only if the ground field has a method `"D"`, too.
- This method overloads the function `D`.
- This method must not be called for inseparable extensions; note that MuPAD cannot check whether an extension is separable.
- See Example 3.

Method `diff`: partial differentiation

```
diff(dom a, identifier x1, ...)
```

- This method computes $\frac{\partial a}{\partial x_1}$.
- Differentiation is defined to be the continuation of differentiation of the ground field; this method exists only if the ground field has a method `"diff"`, too.
- Differentiation is not possible in inseparable extensions.
- This method overloads the function `diff`.
- This method must not be called for inseparable extensions; note that MuPAD cannot check whether an extension is separable.
- See Example 3.

Method `random`: random element of the field

```
random()
```

- This method returns a random element of the field.
- The `random` method of the ground field is used to generate coefficients of a random polynomial of the ground field; the residue class of that polynomial is the return value. Hence the probability distribution of the elements returned depends on that of the `random` method of the ground field.

Conversion Methods

Method `convert`: convert into a field element

```
convert(any x)
```

- This method tries to convert `x` into a field element.
- If the conversion fails, then `FAIL` is returned.

Method `convert_to`: convert a field element into another type

```
convert_to(dom a, domain T)
```

- This method converts `a` into an element of `T`, or returns `FAIL` if the conversion is impossible.
- Field elements can be converted to polynomials or expressions. Field elements represented by constant polynomials can also be converted to the same types as the elements of the ground field; in particular, they can be converted to elements of the ground field.

Method `expr`: convert an element of the field into an expression

`expr(`*dom* `a)`

- This method converts the polynomial representing `a` into an expression.
- This method overloads the function `expr`.

Example 1. We adjoin a cubic root `alpha` of 2 to the rationals.

```
>> G := Dom::AlgebraicExtension(Dom::Rational, alpha^3 = 2)

                                                  3
  Dom::AlgebraicExtension(Dom::Rational, alpha  - 2 = 0, alpha)
```

The third power of a cubic root of 2 equals 2, of course.

```
>> G(alpha)^3

                                    2
```

The trace of α is zero:

```
>> G::conjTrace(G(alpha))

                                    0
```

You can also create random elements:

```
>> G::random()

                                          2
                  - 65 alpha - 814 alpha  + 824
```

Example 2. The ground field may be an algebraic extension itself. In this way, it is possible to construct a tower of fields. In the following example, an algebraic extension is defined using a primitive element `alpha`, and the primitive element `beta` of a further extension is defined in terms of `alpha`. In such cases, when a minimal equation contains more than one identifier, a third argument to `Dom::AlgebraicExtension` must be explicitly given.

```
>> F := Dom::AlgebraicExtension(Dom::Rational, alpha^2 = 2):
   G := Dom::AlgebraicExtension(F, bet^2 + bet = alpha, bet)

 Dom::AlgebraicExtension(Dom::AlgebraicExtension(Dom::Rational,

         2                                        2
    alpha  - 2 = 0, alpha), bet - alpha + bet  = 0, bet)
```

Example 3. We want to define an extension of the field of fractions of the ring of bivariate polynomials over the rationals.

```
>> P:= Dom::DistributedPolynomial([x, y], Dom::Rational):
   F:= Dom::Fraction(P):
   K:= Dom::AlgebraicExtension(F, alpha^2 = x, alpha)

 Dom::AlgebraicExtension(Dom::Fraction(

    Dom::DistributedPolynomial([x, y], Dom::Rational, LexOrder))

           2
    , alpha  - x = 0, alpha)
```

Now $K = Q[sqrt(x), y]$. Of course, the square root function has the usual derivative; note that $1/sqrt(x)$ can be expressed as α/x:

```
>> diff(K(alpha), x)
```

$$\frac{\alpha}{2x}$$

On the other hand, the derivative of $sqrt(x)$ with respect to y is zero, of course:

```
>>  diff(K(alpha), y)
```

$$0$$

We must not use `D` here. This works only if we start our construction with a ring of univariate polynomials:

```
>> P:= Dom::DistributedPolynomial([x], Dom::Rational):
   F:= Dom::Fraction(P):
   K:= Dom::AlgebraicExtension(F, alpha^2 = x, alpha):
   D(K(alpha))
```

$$\frac{\alpha}{2x}$$

Super-Domain: `Dom::BaseDomain`

Axioms

if **`F::hasProp(Ax::canonicalRep)`** **then**
`Ax::canonicalRep`

else
`Ax::normalRep`

Changes:

- An additional method `minimalPolynomial` is now available.
- Methods `diff` and `D` are now available for (partial) differential fields if such methods exist in the ground field.

`Dom::ArithmeticalExpression` – the domains of arithmetical expressions

`Dom::ArithmeticalExpression` creates the domain of arithmetical expressions built up by the system functions and operators like `+` and `*`.

Creating Elements:

- `Dom::ArithmeticalExpression(x)`

Parameters:

`x` — an arithmetical expression

Categories:

`Cat::BaseCategory`

Related Domains: `Dom::Expression`

Details:

- `Dom::ArithmeticalExpression` is a façade domain of arithmetical expressions built up by the system functions and operators like `+` and `*`.
- This domain has almost no algebraic structure because unqualified expressions have no normal form. (For example, there are rational expressions for zero which are not normalized to `0`.) The main purpose of `Dom::ArithmeticalExpression` is to provide implementations for methods used by façade sub-domains like `Dom::Integer` which are represented by a subset of the arithmetical expressions.
- Elements of `Dom::ArithmeticalExpression` are usually not created explicitly. However, if one creates elements using the usual syntax, the input is converted to an expression using `expr`, then it is checked whether the result is an arithmetical expression.

Entries:

`key`	The name of this domain.
`one`	The neutral element w.r.t. `"_mult"`: the constant `1`.
`zero`	The neutral element w.r.t. `"_plus"`: the constant `0`.

Mathematical Methods

Method `_divide`: divides arithmetical expressions

`_divide(`*`dom`* `f, `*`dom`* `g)`

- Returns the arithmetical expression `f/g`.
- This method overloads the function `_divide`.
- For details, please see `_divide`.

Method `_invert`: inverts an arithmetical expression

`_invert(`*`dom`* `f)`

- Returns the arithmetical expression `1/f`.
- This method overloads the function `_invert`.
- For details, please see `_invert`.

Method `_mult`: multiplies arithmetical expressions

`_mult(<f,g, ...> )`

- Multiplies an arbitrary number of arithmetical expressions and returns the (simplified) arithmetical expression `f*g*...`
- This method overloads the function `_mult`.
- For details, please see `_mult`.

Method `_negate`: negates an arithmetical expression

`_negate(`*`dom`* `f)`

- Returns the arithmetical expression `-f`.
- This method overloads the function `_negate`.
- For details, please see `_negate`.

Method `_plus`: adds arithmetical expressions

`_plus(<f,g, ...> )`

- Adds an arbitrary number of arithmetical expressions and returns the (simplified) arithmetical expression `f+g+...`
- This method overloads the function `_plus`.
- For details, please see `_plus`.

Method `_power`: power operator

`_power(`*`dom`* `f, `*`dom`* `g)`

- Returns the arithmetical expression `f^g`.
- This method overloads the function `_power`.
- For details, please see `_power`.

Method `_subtract`: subtracts an arithmetical expression

`_subtract(`*`dom`* `f, `*`dom`* `g)`

- Returns the arithmetical expression `f-g`.
- For details, please see `_subtract`.

Method `D`: differential operator for functions

`D(`*`function`* `f)`

`D(`*`list of nonnegative integers`* `[n1,...], `*`function`* `f)`

- `D(f)` computes the derivative of `f` and returns a function or functional expression which may contain unevaluated calls of `D`.
- This method overloads the function `D`.
- For details, please see `D`.

Method `diff`: differentiates an arithmetical expression

`diff(`*`dom`* `f <,x, ...>)`

- Differentiates `f` with respect to the given sequence of variables in the given order and returns an arithmetical expression. If `x` is not a variable, `f` will be evaluated and returned without differentiation.
- This method overloads the function `diff`.
- For details, please see `diff`.

Method `intmult`: multiplies an arithmetical expression with an integer

```
intmult(dom f, integer n)
```

- Returns the arithmetical expression `f*n`.
- This method overloads the function `_mult`.
- For details, please see `_mult`.

Method `iszero`: test for zero

```
iszero(dom f)
```

- Tests whether `f` is zero and returns `TRUE` or `FALSE`.
- This method overloads the function `iszero`.
- For details, please see `iszero`.

Method `max`: maximum of numbers

```
max(dom x <,y, ...> )
```

- Calculates the maximum of numerical elements. If one of the arguments cannot be evaluated to a number, then the function call with all non–numerical and the minimum of the numerical arguments is returned.
- All numerical arguments must be real.
- This method overloads the function `max`.
- For details, please see `max`.

Method `min`: minimum of numbers

```
min(dom x <,y, ...> )
```

- Calculates the minimum of numerical elements. If one of the arguments cannot be evaluated to a number, then the function call with all non–numerical and the minimum of the numerical arguments is returned.
- All numerical arguments must be real.
- This method overloads the function `min`.
- For details, please see `min`.

Method `norm`: norm of an arithmetical expression

```
norm(dom f)
```

- Computes the norm of `f` as the absolute value of `f`.
- This method overloads the function `abs`.
- For details, please see `abs`.

Conversion Methods

Method `convert`: check for being an arithmetical expression

`convert(`*`any`* `x)`

- Tests whether `x` is an arithmetical expression. If yes, `x` is returned; otherwise the result is `FAIL`.

Example 1. For brevity, we will use `AE` as a shorthand notation for `Dom::ArithmeticalExpressio`

```
>> AE := Dom::ArithmeticalExpression

                    Dom::ArithmeticalExpression
```

An element of this domain can *not* be created as follows:

```
>> e := AE(2*sin(x) + f(x)/y)

                                  f(x)
                        2 sin(x) + ----
                                   y
```

Since `Dom::ArithmeticalExpression` is a façade domain, `e` is not a domain element, but an expression:

```
>> domtype(e)

                                DOM_EXPR
```

The fact that no error was returned yields the information that `e` is an arithmetical expression. This can also be checked as follows:

```
>> testtype(e,AE)

                                  TRUE
```

In contrast to its super-domain `Dom::Expression`, this domain only allows elements which are valid arguments for the arithmetical functions, thus the following yields an error:

```
>> AE([a, b])

 Error: illegal arguments [Dom::ArithmeticalExpression::new]
```

Super-Domain: `Dom::Expression`

Axioms

`Ax::systemRep`

Changes:

- No changes.

`Dom::BaseDomain` – the root of the domain hierarchy

`Dom::BaseDomain` is the root of the domain hierarchy of the `Dom` package. Every domain of the package inherits from it.

Domain:

- `Dom::BaseDomain`

Details:

- `Dom::BaseDomain` is the root of the domain hierarchy as defined by the `Dom` package. Every domain of the package inherits from it.
- The only purpose of `Dom::BaseDomain` is to supply all domains of the package with some basic methods like `"hasProp"`. Elements of `Dom::BaseDomain` cannot be created.
- Unlike other super-domains this domain does not impose any restrictions on the representation of the elements of its sub-domains. Thus it may be a super-domain for any domain created by a domain constructor.

Categories:

`Cat::BaseCategory`

Entries:

`create_dom` This domain entry is used to revive the domain when it is read from a binary MCode stream.

If this entry is present it is written to the MCode stream instead of the contents of the domain. When the stream is read it is used to create the domain.

If this entry does not exist all entries of the domain are written to the stream and read in later to create the domain.

`Dom::BaseDomain` defines `"create_dom"` to have the same value as the key of the domain, as stored in the entry `"key"`. All domains

of the Dom package inherit this entry, thus they must be created by the reader of the MCode stream by evaluating the expression stored in the key.

Mathematical Methods

Method `equal`: test for mathematical equality

```
equal(dom x, dom y)
```

- This method must return TRUE if it can decide that x is equal to y in the mathematical sense imposed by this domain. It must return FALSE if it can decide that x is not equal to y mathematically. If the method cannot decide the equality it must return UNKNOWN, see Cat::BaseCategory.
- If this domain has the axiom Ax::canonicalRep, which implies that two domain elements are mathematically equal if and only if they are structurally equal, the kernel function _equal is used to decide the equality. In this case UNKNOWN is never returned.
- If the axiom Ax::canonicalRep does not hold the method will return TRUE if x and y are structurally equal (in the sense of the function _equal) and UNKNOWN otherwise.

Conversion Methods

Method `convert_to`: convert element

```
convert_to(dom x, type T)
```

- This method may be used to convert elements of this domain to a given type T. It returns FAIL if a conversion is not possible.
- The implementation provided here can convert x to an element of this domain (the trivial case) or to an element of Dom::Expression (by using the method "expr", see Cat::BaseCategory).

Method `TeX`: generate TeX output

```
TeX(dom x)
```

- Returns a TeX-formatted string for x.
- The default implementation provided here converts x into an expression using the method "expr" and then uses the function generate::TeX to convert the expression.

Access Methods

Method `allAxioms`: return all axioms

`allAxioms()`

- Returns a set containing all axioms holding for this domain, as stated explicitly or in the inherited domains or categories.

Method `allCategories`: return all categories

`allCategories()`

- Returns a list containing all all categories of this domain. The order of the categories in the list is the order which is used to search for entries of the categories.

Method `allEntries`: return the names of all entries

`allEntries()`

- Returns a set containing the names of all entries of this domain.

Method `allSuperDomains`: return all super-domains

`allSuperDomains()`

- Returns a list containing all super-domains of this domain. The order of the domains in the list is given by the domain hierarchy: The first domain is the direct super-domain, the second the direct super-domain of the first and so on.
- The last, most general, super-domain of all domains of the `Dom` package is `Dom::BaseDomain`.

Method `getAxioms`: return axioms stated in the constructor

`getAxioms()`

- Returns a set containing the axioms stated directly for this domain.

Method `getCategories`: return categories stated in the constructor

`getCategories()`

- Returns a list containing the categories of this domain which are are listed directly by the domain constructor.

Method `getSuperDomain`: return super-domain stated in the constructor

```
getSuperDomain()
```

- Returns the direct super-domain of this domain as given by the domain constructor.

Method `hasProp`: test for a certain property

```
hasProp(DOM_DOMAIN d)
```

- Tests if the domain `d` is this domain or a super-domain of this domain.

```
hasProp(DomainConstructor dc)
```

- Tests if this domain or a super-domain of it was defined by the domain constructor `dc`.

```
hasProp(Axiom a)
```

- Tests if this domain has the axiom `a`.

```
hasProp(AxiomConstructor ac)
```

- Tests if an axiom of this domain was defined by the axiom constructor `ac`.

```
hasProp(Category c)
```

- Tests if this domain has the category `c`.

```
hasProp(CategoryConstructor cc)
```

- Tests if a category of this domain was defined by the category constructor `cc`.

Method `info`: prints short information about this domain

```
info()
```

- This method overloads the function `info`.
- It prints out the super-domains, categories, axioms and entry names of this domain.
- If an entry `"info_str"`, which must be a string, is defined for this domain it is used to print the header line.

Method `printMethods`: prints out methods

`printMethods(<`*`function`* `sort,> `*`Table`*`)`

- Sorts the names of all entries of this domain using the function `sort` and then prints them. The entries are grouped according to the domains and categories defining them and printed out in a tabular form.
- If no sorting function is given, `sort` is used as default.

`printMethods(<`*`function`* `sort,> `*`Tree`*`)`

- Similar as above, only that the names of the entries are inserted into a tree, an element of the domain `adt::Tree`. The tree is both printed out and returned by the method.

`printMethods(<`*`function`* `sort>)`

- Does the same as `dom::printMethods(sort, Table)`.

Method `subs`: avoid substitution

`subs(`*`dom`* `x, ...)`

- This method overloads the function `subs`. The implementation provided here simply returns `x` in any case, so it avoids unwanted substitutions inside of domain elements.
- Sub-domains should provide a new implementation of this method with sensible semantics if possible.

Method `subsex`: avoid extended substitution

`subsex(`*`dom`* `x, ...)`

- This method overloads the function `subsex`. The implementation provided here simply returns `x` in any case, so it avoids unwanted substitutions inside of domain elements.
- Sub-domains should provide a new implementation of this method with sensible semantics if possible.

Method `undefinedEntries`: return missing entries

`undefinedEntries()`

- Returns a set containing the names of all those entries of this domain that are missing.
- An entry is missing if it should have a definition according to a category of the domain, but the definition is not present.

Method `whichEntry`: return the domain or category implementing an entry

```
whichEntry(e)
```

- Returns the domain or category which implements the entry with name `e`.
- `FAIL` is returned if no entry with the given name is defined for this domain.

Changes:

- New methods: `"create_dom"` and `"printMethods"`.

`Dom::Complex` – the field of complex numbers

`Dom::Complex` is the field of complex numbers represented by elements of the domains `DOM_INT`, `DOM_RAT`, `DOM_FLOAT`, `DOM_COMPLEX` and `DOM_EXPR`.

Creating Elements:

- `Dom::Complex(x)`

Parameters:

`x` — An expression of type `DOM_INT`, `DOM_RAT`, `DOM_FLOAT`, `DOM_COMPLEX`. An expression of type `DOM_EXPR` is also possible if it is of type `Type::Arithmetical` and if it contains only indeterminates which are of type `Type::ConstantIdents` or if it contains no indeterminates.

Categories:

`Cat::DifferentialRing`, `Cat::Field`

Related Domains: `Dom::Float`, `Dom::Integer`, `Dom::Numerical`, `Dom::Rational`, `Dom::Real`

Details:

- `Dom::Complex` is the domain of complex constants represented by expressions of type `DOM_INT`, `DOM_RAT`, `DOM_FLOAT` or `DOM_COMPLEX`. An expression of type `DOM_EXPR` is considered a complex number if it is of type `Type::Arithmetical` and if it contains only indeterminates which are of type `Type::ConstantIdents` or if it contains no indeterminates, cf. example 2.

- `Dom::Complex` is of category `Cat::Field` due to pragmatism. This domain actually is not a field because `bool(1.0 = float(3) / float(3))` returns `FALSE`, for example.
- Elements of `Dom::Complex` are usually not created explicitly. However, if one creates elements using the usual syntax, it is checked whether the input expression can be converted to a number. This means `Dom::Complex` is a facade domain which creates elements of domain type `DOM_INT`, `DOM_RAT`, `DOM_FLOAT`, `DOM_COMPLEX` or `DOM_EXPR`.
- `Dom::Complex` has no normal representation, because `0` and `0.0` both represent the zero.
- Viewed as a differential ring, `Dom::Complex` is trivial. It only contains constants.
- `Dom::Complex` has the domain `Dom::BaseDomain` as its super domain, i.e., it inherits each method which is defined by `Dom::BaseDomain` and not re-implemented by `Dom::Complex`. Methods described below are re-implemented by `Dom::Complex`.

Entries:

`characteristic`	the characteristic of this field is 0.
`one`	the unit element; it equals `1`.
`zero`	The zero element; it equals `0`.

Mathematical Methods

Method `_divide`: divide numbers

```
_divide(dom x, dom y)
```

- Behaves like the function `_divide`.

Method `_invert`: invert numbers

```
_invert(dom x)
```

- Behaves like the function `_invert`.

Method `_mult`: multiplies numbers

```
_mult(dom x, dom y, ...)
```

- Behaves like the function `_mult`.

Method `_negate`: negate numbers

`_negate(`*dom* `x)`

- Behaves like the function `_negate`.

Method `_plus`: add numbers

`_plus(`*dom* `x, `*dom* `y, ...)`

- Behaves like the function `_plus`.

Method `_power`: power operator

`_power(`*dom* `x, `*dom* `y)`

- Behaves like the function `_power`.

Method `_unequal`: inequalities

`_unequal(`*dom* `x, `*dom* `y)`

- Behaves like the function `_unequal`.

Method `conjugate`: conversion to a basic type

`conjugate(`*dom* `x)`

- Behaves like the function `conjugate`.

Method `D`: differential operator

`D(`*dom* `x)`

- This method returns 0.

Method `diff`: differentiates

`diff(`*dom* `z, <, `*any* `x, ...> )`

- This method returns `z` if it is called with only one argument. Otherwise it returns 0.

Method `equal`: equations

`equal(`*dom* `x, `*dom* `y)`

- Behaves like the function `_equal`.

Method `expr`: conversion to a basic type

`expr(`*`dom`* `x)`

- Behaves like the function `expr`.

Method `iszero`: zero test

`iszero(`*`dom`* `x)`

- Behaves like the function `iszero`.

Method `norm`: the absolute value of a number

`norm(`*`dom`* `x)`

- This method returns $|x|$.

Method `random`: random number generation

`random()`

- This method returns a randomly generated complex number where the real part and the imaginary part are positive integers between 0 and $10^{12} - 1$ generated by `random`.

`random(`*`integer`* `n)`

- This method returns a random number generator which creates complex random numbers where the real parts and the imaginary parts are positive integers between 0 and $n - 1$.

`random(`*`integer`* `m..`*`integer`* `n)`

- This method returns a random number generator which creates complex random numbers where the real parts and the imaginary parts are positive integers between m and n.

Method `unequal`: inequalities

`unequal(`*`dom`* `x, `*`dom`* `y)`

- Behaves like the function `_unequal`.

Conversion Methods

Method `convert`: conversion into this domain

`convert(`*`any`* `x)`

- This method tries to convert `x` to a number of type `Dom::Complex`. Currently this method can convert elements of type `DOM_INT`, `DOM_RAT`, `DOM_FLOAT`, `DOM_COMPLEX`. It also can convert constant identifier like `PI`, `EULER` and `CATALAN`.
- An arithmetical expression can be converted if it only contains subexpression of the types just mentioned.
- If the conversion fails, `FAIL` is returned.

Method `convert_to`: conversion to other domains

`convert_to(`*`dom`* `x,` *`any`* `T)`

- This method tries to convert the number `x` to an element of type `T`, or if `T` is not a domain, to the domain type of `T`.
- If the conversion fails, `FAIL` is returned.
- The following domains are allowed for `T`: `DOM_INT`, `Dom::Integer`, `DOM_RAT`, `Dom::Rational`, `DOM_FLOAT`, `Dom::Float`, `Dom::Numerical`, `DOM_COMPLEX` and `DOM_EXPR`.

Method `normal`: normal form of objects

`normal(`*`dom`* `x)`

- Behaves like the function `normal`.

Example 1. Creating some complex numbers using `Dom::Complex`:

```
>> Dom::Complex(2/3)

                                  2/3

>> Dom::Complex(2/3 + 4*I)

                              2/3 + 4 I
```

Example 2. It's also possible to use expressions or constants for creating an element of `Dom::Complex`:

```
>> Dom::Complex(PI)

                                  PI

>> Dom::Complex(sin(2))

                               sin(2)

>> Dom::Complex(sin(2/3*I) + 3)

                           I sinh(2/3) + 3
```

If the expression cannot be converted to an element of `Dom::Complex` we will get an error message:

```
>> Dom::Complex(sin(x))

 Error: illegal arguments [Dom::Complex::new]
```

Super-Domain: `Dom::ArithmeticalExpression`

Axioms

`Ax::systemRep`, `Ax::efficientOperation("_divide")`,
`Ax::efficientOperation("_mult")`,
`Ax::efficientOperation("_invert")`

Changes:

- No changes.

`Dom::DihedralGroup` – dihedral groups

`Dom::DihedralGroup(n)` creates the group of all congruent mappings of the plane that induce a bijective mapping of the set of corners of a regular n-angle to itself.

Domain:

- `Dom::DihedralGroup(n)`

Parameters:

`n` — positive integer

`Dom::DihedralGroup(n)([a,b])` represents the group element "t^a carried out after r^b", where r is a rotation that maps each corner to its left neighbor, and t is a reflection w.r.t. some fixed central diagonal.

Creating Elements:

- `Dom::DihedralGroupn(l)`

Parameters:

`l` — list or array of two integers

Categories:

`Cat::Group`

Related Domains: `Dom::PermutationGroup`

Entries:

`size`	the number of elements, which equals $2n$.
`one`	the mapping leaving each point fixed.

Mathematical Methods

Method `_mult`: functional composition of elements

`_mult(`*`dom`* `a, ...)`

- The product of elements of `Dom::DihedralGroup` is defined as their functional composition, with the factors applied from right to left.
- This method overloads the kernel function `_mult`.

Method `_invert`: inverse of an element

`_invert(`*`dom`* `a)`

- The inverse of `a` is defined to be the mapping that sends every corner to its pre-image under `a`. (This agrees with the usual notion of the inverse of a bijective mapping.)
- This method overloads the kernel function `_invert`.

Method `_power`: power of an element

`_power(`*`dom`* `a,` *`integer`* `n)`

- This method computes a^n.
- It overloads the kernel function `_power`.

Method `order`: order of a group element

`order(`*`dom`* `a)`

- This method returns the smallest positive integer k for which $a^k = 1$.

Method `random`: random element

`random()`

- This method returns a random element of the group; the results are uniformly distributed.

Conversion Methods

Method `expr`: convert group element to list

`expr(`*`dom`* `a)`

- This method returns a list of two elements; the meaning is the same as in the case of creating elements of `Dom::DihedralGroup`.

Method `TeX`: TeX output of a group element

`TeX(`*`dom`* `a)`

- The element `a` is returned as a TEXstring generated from its list representation. This avoids using fixed names for the generators, as there is no standard for them in the literature.

Example 1. Define the group D_6, i.e., the group of congruence mappings of the hexagon:

```
>> G := Dom::DihedralGroup(6)
```

```
                        Dom::DihedralGroup(6)
```

Then elements may be created as follows:

```
>> a := G([7, 19]);
```

```
                                [1, 1]
```

This means that 19 rotations—mapping each corner to its left neighbor—and 7 reflections have the same effect as one operation of either type.

Super-Domain: `Dom::BaseDomain`

Axioms

`Ax::canonicalRep`

Changes:

- No changes.

`Dom::DistributedPolynomial` – the domains of distributed polynomials

`Dom::DistributedPolynomial(Vars, R, ..)` creates the domain of polynomials in the variables of the list `Vars` over the commutative ring `R` in distributed representation.

Domain:

- `Dom::DistributedPolynomial(<Vars <, R <, Order>>>)`

Parameters:

`Vars` — a list of indeterminates. Default is `[]` (the empty list, indicating "arbitrary indeterminates").

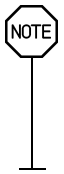
`R` — a commutative ring, i.e., a domain of category `Cat::CommutativeRing`. Default is `Dom::ExpressionField(normal)`.

`Order` — a monomial ordering, i.e., one of the predefined orderings `LexOrder`, `DegreeOrder` or `DegInvLexOrder` or any object of type `Dom::MonomOrdering`. Default is `LexOrder`.

Details:

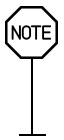
- `Dom::DistributedPolynomial(Vars, R, Order)` creates a domain of polynomials in the variables of the list `Vars` over a domain of category `Cat::CommutativeRing` in sparse distributed representation with respect to the monomial ordering `Order`.
- If `Dom::DistributedPolynomial` is called without any argument, a polynomial domain in arbitrarily many indeterminates over the domain `Dom::ExpressionField(normal)` with respect to the lexicographic monomial ordering is created.

- If `Dom::DistributedPolynomial` is called only with the variable list `Vars` as argument, the polynomial domain in the variable list `Vars` over the domain `Dom::ExpressionField(normal)` with respect to the lexicographic monomial ordering is created.
- Only commutative coefficient rings of type `DOM_DOMAIN` are allowed which inherit from `Dom::BaseDomain`. If `R` is of type `DOM_DOMAIN` but does not inherit from `Dom::BaseDomain`, the domain `Dom::ExpressionField(normal)` will be used instead.



- `Dom::DistributedPolynomial` accepts expressions as indeterminates, similar to the kernel domain `DOM_POLY`. Hence, for example, `[x,cos(x)]` is a valid variable list.
- If the variable list `Vars` is the empty list (`[]`), a polynomial domain in arbitrarily many indeterminates is created. In this case, when creating new elements from polynomials or polynomial expressions, the system function `indets` is first called to get the variables and then the polynomial is created with respect to these variables. Hence, in this case only identifiers can be valid indeterminates, because `indets` returns only identifiers.
- It is not allowed to create polynomial domains in arbitrarily many indeterminates over another polynomial domain of category `Cat::Polynomial`, but it is possible to create multivariate polynomial domains with a given list of variables over any polynomial domain.
- `Dom::DistributedPolynomial` represents polynomials over arbitrary commutative rings. It is intended as a basic domain for distributed polynomials from which it is easy to create new distributed polynomial domains.

  All usual algebraic and arithmetical polynomial operations are implemented, including Gröbner basis computation.
- It is highly recommend to use only coefficient rings with unique zero representation. Otherwise it can happen that, e.g., a polynomial division will not terminate or a wrong degree will be returned.



- Please note that for reasons of efficiency not all methods check their arguments, not even at the interactive level. In particular this is true for many access methods, converting methods and technical methods.

Creating Elements:

- `Dom::DistributedPolynomial(Vars, R, Order)(p)`
- `Dom::DistributedPolynomial(Vars, R, Order)(lm)`

- `Dom::DistributedPolynomial(Vars, R, Order)(lm,v)`

Parameters:

`p` — a polynomial or a polynomial expression.
`lm` — list of monomials, which are represented as lists containing the coefficients together with the exponents or exponent vectors.
`v` — list of indeterminates. This parameter is only valid for `Vars` = `[]`.

Categories:

if `Vars` has a single variable, then
`Cat::UnivariatePolynomial(R)`

else
`Cat::Polynomial(R)`

Related Domains: `Dom::Polynomial`, `Dom::MultivariatePolynomial`, `Dom::UnivariatePolynomial`

Entries:

`characteristic` The characteristic of this domain.

`coeffRing` The coefficient ring of this domain as defined by the parameter `R`.

`key` The name of the created domain.

`one` The neutral element w.r.t. `"_mult"`.

`ordering` The monomial order as defined by the parameter `Order`.

`variables` The list of variables as defined by the parameter `Vars`.

`zero` The neutral element w.r.t. `"_plus"`.

Mathematical Methods

Method `_divide`: exact polynomial division

`_divide(`*`dom`* `a, `*`dom`* `b)`
`_divide(`*`dom`* `a, `*`R`* `b)`
`_divide(`*`dom`* `a, `*`DOM_INT`* `b)`

- Divides `a` by `b` and returns the divisor, if `a` is divisible by `b` otherwise `FAIL`.
- It overloads the function `_divide` for polynomials, i.e., one may use it either in the form `a / b`, or in functional form `_divide(a, b)`.

- This method only exists if R is an integral domain, i.e., a domain of category Cat::IntegralDomain.

Method _invert: inverse of an element

_invert(*dom* a)

- Returns the inverse of a if it exists, otherwise FAIL.

Method _mult: multiplies polynomials and coefficient ring elements

_mult(<a,b, ...>)

- Multiplies an arbitrary number of polynomials of this domain, elements of the coefficient ring R and elements of type DOM_INT and returns an element of this domain. If any element of a different domain occurs as an argument, the method from that domain is called and the result of that call is returned. If this call fails, FAIL is returned.
- This method overloads the function _mult for polynomials, i.e., one may use it either in the form a * b * ... or in functional notation _mult(a, b, ...).

Method _negate: negates a polynomial

_negate(*dom* a)

- Negates a, i.e., multiplies a by -1.
- This method overloads the function _negate for polynomials, i.e., one may use it either in the form -a or in functional notation _negate(a).

Method _plus: adds polynomials and coefficent ring elements

_plus(<a,b, ...>)

- Adds an arbitrary number of polynomials of this domain together with elements of the coefficient ring R and elements of type DOM_INT and returns an element of this domain. When any other element of a different domain occurs as argument, the method from that domain is called and an element of that domain is returned. If this call fails, FAIL is returned.
- This method overloads the function _plus for polynomials, i.e., one may use it either in the form a + b + ... or in functional notation _plus(a, b, ...).

Method `_power`: nth power of a polynomial

```
_power(dom a, NonNegativeInteger n)
```

- Returns the `n`-th power of `a`, i.e., multiplies `a` `n` times by itself.
- This method overloads the function `_power` for polynomials, i.e., one may use it either in the form `a^n` or in functional notation `_power(a,n)`.

Method `_subtract`: subtracts a polynomial or a coefficient ring element

```
_subtract(a, b)
```

- Subtracts `a` by `b`, whereby both arguments can be either polynomials of this domain or elements of the coefficient ring `R` or elements of type `DOM_INT` and returns an element of this domain. If not possible, an error message will occur.
- This method overloads the function `_subtract` for polynomials, i.e., one may use it either in the form `a - b` or in functional notation `_subtract(a, b)`.

Method `content`: content of a polynomial

```
content(dom a)
```

- computes the content of `a`, i.e., the gcd of all coefficients.
- This method only exists if `R` is a domain of category `Cat::GcdDomain`.

Method `D`: differential operator for polynomials

```
D(dom a)
D(list of positive integers l, dom a)
```

- This method overloads the function `polylib::Dpoly` for polynomials and is simply another name for the method `"Dpoly"`.

Method `Dpoly`: differential operator for polynomials

```
Dpoly(dom a)
Dpoly(list of positive integers l, dom a)
```

- `Dpoly(a)` computes the derivative of `a`, if `a` has exactly one indeterminate.
- `Dpoly(l,a)` computes the partial derivative of `a` with respect to `l`. For details see `polylib::Dpoly`.
- This method overloads the function `polylib::Dpoly` for polynomials.

Method `decompose`: functional decomposition of a polynomial

`decompose(`*`dom`* `a <,` *`indeterminate`* `var>)`

- Returns a sequence of of polynomials of this domain `p1, ..., pn` such that `a` = `p1(...pn(var)...)`.
- If `a` is a polynomial in only one variable, the second argument is not necessary.
- This method overloads the function `polylib::decompose` for polynomials.

Method `diff`: differentiates a polynomial

`diff(`*`dom`* `a,` *`sequence of indeterminates`* `varseq)`

- Returns the partial derivative of `a` with respect to the sequence of indeterminates `varseq`.
- If `varseq` is an empty sequence, `a` is returned unchanged.
- If in `varseq` an expression occurs which is not a variable of `a`, the zero polynomial is returned.
- This method overloads the function `diff` for polynomials.

Method `dimension`: dimension of affine variety

`dimension(`*`list of dom`* `ais <,` *`monomial ordering`* `ord>)`

`dimension(`*`set of dom`* `ais <,` *`monomial ordering`* `ord>)`

- Computes the dimension of the affine variety generated by the polynomials of `ais` with respect to the monomial ordering `ord`, if explicitly given, otherwise `Order` will be used instead.
- This method is merely an interface for the function `groebner::dimension`.
- This method only exists if `R` is a field, i.e., a domain of category `Cat::Field` and `Vars` is not the empty list.

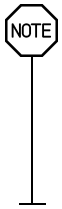
Method `divide`: divides polynomials

`divide(`*`dom`* `a,` *`dom`* `b <,` *`Quo, Rem or Exact`* `opt>)`

`divide(`*`dom`* `a,` *`dom`* `b,` *`indeterminate`* `var <,` *`Quo, Rem or Exact`* `opt>)`

- Divides `a` by `b`, whereby both polynomials are univariate in a common variable or with respect to the given indeterminate `var`. Note that if `Vars` has a single variable the second call leads to an error. If no indeterminate is given the main variable of the first polynomial will be used.

- If no option is given, the quotient `s` and the remainder `r` are computed such that `a` = `s*b + r` and the degree of `r` in the relevant indeterminate is smaller than that of `b`. The sequence consisting of `s, r` is returned, otherwise `FAIL`.
- If the option `Quo` is given, only the quotient `s` is returned.
- If the option `Rem` is given, only the remainder `r` is returned.
- If the option `Exact` is given, only the quotient `s` is returned, in case the remainder is zero, otherwise `FAIL`.
- `divide(a,b,Exact)` divides the multivariate polynomial `a` by `b`. If `a` cannot be divided by `b`, the method returns `FAIL`.
- This method overloads the function `divide` for polynomials.
- This method only exists if `R` is a field, i.e., a domain of category `Cat::Field` and either this domain is of category `Cat::UnivariatePolynomial(R)` or `R` has characteristic zero (`R::characteristic` = 0). If the first pair of conditions is true then the first call is valid otherwise the second one.



Method `evalp`: evaluates a polynomial

`evalp(`*`dom`* `a,` *`sequence of equations`* `var = e)`

- Evaluates the polynomial `a` by substituting the variables `var, ...` by `e, ...`. The values `e, ...` should be elements of the coefficient ring or expressions that could be used as coefficients. The variables are evaluated in the sequence given by the equations using Horner's rule. An element of this domain or an element of the coefficient ring respectively is returned.
- This method overloads the function `evalp` for polynomials.

Method `factor`: factors a polynomial

`factor(`*`dom`* `a)`

- Computes a factorization of the polynomial `a` into irreducible factors such that `a` = `u*a1^e1 ...*an^en` where `u` is the content of `a` and `ai` ($1 \leq i \leq n$) are the irreducible factors of `a` and returns an element of domain type `Factored`. See `factor` for more details.
- This method overloads the function `factor` for polynomials.
- This method only exists if `R` is a domain of category `Cat::Field` or if `R` is the domain `Dom::Integer`.

Method `func_call`: applies expressions to a polynomial

```
func_call(dom a, R e1, ..., R en, <Expr>)
func_call(dom a, dom e1, ..., dom en, <Expr>)
func_call(dom a, expression e1, ..., expression en, Expr)
```

- This method may also be used either in the form `a(e1,..., en, <Expr>)` or in functional notation `func_call(a, e1,..., en, <Expr>)`.
- `a(e1,..., en)` applies the sequence `e1,..., en` of either elements of this domain or elements of `R` with respect to `Vars` (where `n` is the number of variables) to the polynomial `a`. An element of this domain or an element of the coefficient ring respectively is returned.
- `a(e1,..., en, Expr)` applies the sequence of expressions or of elements of this domain or of elements of `R` to the polynomial `a`. With this call `a` is first converted into an expression. Afterwards `e1,..., en` is substituted into this expression with respect to `Vars`. The return value may be any object.
- The number of variables must be equal to the number of applied expressions.
- This method only exists if `Vars` has at least one indeterminate.

Method `gcd`: greatest common divisor of polynomials

```
gcd(dom a, dom b, ...)
```

- Computes a greatest common divisor of the polynomials `a, b, ...`
- This method overloads the function `gcd` for polynomials.
- This method only exists if `R` is a domain of category `Cat::GcdDomain`.

Method `gcdex`: extended Euclidean algorithm for polynomials

```
gcdex(dom a, dom b)
```

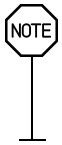
- Applies the extended Euclidean algorithm to compute polynomials `s` and `t` such that `g = s*a+t*b`, where `g` is a greatest common divisor of `a` and `b` and `degree(s) < degree(b)` and `degree(t) < degree(a)`. The sequence of the three polynomials of this domain `g, s, t` is returned.
- This method overloads the function `gcdex` for polynomials. Especially, it only works for coefficient rings described there.
- This method only exists if `R` is a domain of category `Cat::GcdDomain`.

Method `groebner`: reduced Gröbner basis

`groebner(`*`list of dom`* `ais <, `*`monomial ordering`* `ord> <, Reorder>)`

`groebner(`*`set of dom`* `ais <, `*`monomial ordering`* `ord> <, Reorder>)`

- Computes a reduced normalized Gröbner-basis for the ideal generated by the polynomials of `ais` with respect to the monomial ordering `ord` (w.r.t. `Order`, if `ord` is not explicitly given) and returns a list of polynomials of this domain.
- If the option `Reorder` is given, the lexicographical order of variables may change to another one that is likely to decrease the running time.
  Note that this may also cause a change of the returned list, which may now have polynomials over the same coefficient ring `R` but with a possibly re-ordered variable list. Thus, it may contain elements *not* belonging to this domain.



- This method is merely an interface for the function `groebner::gbasis`.
- This method only exists if `R` is a field, i.e., a domain of category `Cat::Field`, and `Vars` is not the empty list.

Method `int`: definite and indefinite integration of a polynomial

`int(`*`dom`* `a <, `*`indeterminate`* `x>)`

`int(`*`dom`* `a <, x=x0..x1>)`

- `int(a,x)` returns the indefinite integral $\int a\,\mathrm{d}x$ as an element either of this domain (if `R` is of category `Cat::Field` or of category `Cat::Algebra(Dom::Rational)` or `Cat::Algebra(Dom::Fraction(Dom::Intege` or as an element of a polynomial domain over `Dom::Fraction(R)` or `FAIL`, if the antiderivative cannot be converted into one of the previously given domains. If `a` has only one variable, the second argument is not necessary.
- `int(a,x=x0..x1)` returns the definite integral $\int_{x0}^{x1} a\,\mathrm{d}x$ or `FAIL`, if the result is not an element of this domain or an element of a polynomial domain over `Dom::Fraction(R)`.
- This method overloads the function `int` for polynomials.

Method `intmult`: multiplies a polynomial with an integer

`intmult(`*`dom`* `a, `*`integer`* `z)`

- Multiplies `a` by `z`.
- This method is more efficient than using polynomial multiplication and is, e.g., necessary for the method `"Dpoly"`.

Method `isone`: test for one

```
isone(dom a)
```

- Tests if `a` is the multiplicative neutral element of this domain and returns `TRUE` if this is the case, otherwise `FALSE`.
- The result can only be valid if the coefficients of `a` are in normal form (i.e., if zero has a unique representation in `R`). Thus, `R` should have at least `Ax::normalRep`.

Method `iszero`: test for zero

```
iszero(dom a)
```

- Tests if `a` is the additive neutral element of this domain and returns `TRUE` if this is the case, otherwise `FALSE`.
- The result can only be valid, if the coefficients of `a` are in normal form (i.e., if zero has a unique representation in `R`). Thus, the coefficient ring `R` should have at least `Ax::normalRep`.

Method `lcm`: least common multiple of polynomials

```
lcm(dom a, dom b, ...)
```

- Computes the least common multiple of the polynomials `a, b, ...`
- This method overloads the function `lcm` for polynomials.
- This method only exists if `R` is a domain of category `Cat::GcdDomain`.

Method `makeIntegral`: makes the coefficients fraction free

```
makeIntegral(dom a)
```

- Multiplies `a` by the `lcm` of all coefficient denominators.
- This method only exists if `R` is a domain of category `Cat::GcdDomain` and `R` has the method `"denom"`.

Method `monic`: normalizes a polynomial

```
monic(dom a)
```

- Normalizes `a`, i.e., the leading coefficient of the resulting polynomial is `R::one`. For this, `a` is divided by its leading coefficient.
- The zero polynomial returns itself.
- This method only exists if `R` is a field, i.e., a domain of category `Cat::Field`.

Method `normalForm`: complete reduction modulo an ideal

`normalForm(`*`dom`* `a, `*`list of dom`* `ais <, `*`monomial ordering`* `ord>)`

`normalForm(`*`dom`* `a, `*`set of dom`* `ais <, `*`monomial ordering`* `ord>)`

- Reduces the polynomial `a` completely modulo all polynomials of `ais` (see `groebner::normalf`) with respect to the monomial ordering `ord` (i.e., w.r.t. `Order`, if `ord` is not explicitly given).
  A reduced form of `a` always exists, but need not be unique; if `ais` form a Gröbner basis, it is unique.
- This method is merely an interface for the function `groebner::normalf`.
- This method only exists if `R` is a field, i.e., a domain of category `Cat::Field`, and `Vars` is not the empty list.

Method `numericSolve`: numerical zeros of polynomials

`numericSolve(`*`dom`* `a <, `*`indeterminate`* `var> <, options>)`

`numericSolve(`*`dom`* `a <, `*`list or set of indeterminates`* `vars> <, options>)`

`numericSolve(`*`list or set of dom`* `ais <, `*`indeterminate`* `var> <, options>)`

`numericSolve(`*`list or set of dom`* `ais <, `*`list or set of indeterminates`* `vars> <, options>)`

- `numericSolve(a, ..)` tries to find the zeros of `a` numerically. It is possible to control the behavior of `numericSolve` by passing a variable or a set or list of variables together with options. For details see the function `numeric::solve`).
- `numericSolve(ais, ..)` tries to find the zeros of the polynomial system `ais` numerically, with the exact behavior depending on further arguments. For details see the function `numeric::solve`).
- All coefficients must be convertible into the basic domain `DOM_EXPR`, since in a precomputation step all polynomials of this domain are converted into the basic polynomial domain `DOM_POLY` over `DOM_EXPR`.
- For a detailed description of possible return values and options see function `numeric::solve`.
- This method overloads the function `numeric::solve`.

Method `pdioe`: solves polynomial Diophantine equations

`pdioe(`*`dom`* `a, `*`dom`* `b, `*`dom`* `c)`

- Returns a sequence of two polynomials `u` and `v` that satisfy the equation `a*u + b*v = c` or `FAIL`, if this equation is not solvable.
- This method overloads the function `solvelib::pdioe`.
- This method only exists if `R` is a field, i.e., a domain of category `Cat::Field` and `Vars` consists of a single variable.

Method `pdivide`: pseudo-division of polynomials

`pdivide(`*`dom`* `a, `*`dom`* `b <, `*`Quo or Rem`* `opt>)`

- Computes the pseudo-division of `a` by `b` and returns the sequence consisting of an element `p` of `R` and two polynomials `q` and `r` of this domain satisfying `p*a = q*b+r`, where `p = lcoeff(b)^(degree(a)-degree(b)+1)`.
- If the option `Quo` is given, only the pseudo-quotient `q` is returned.
- If the option `Rem` is given, only the pseudo-remainder `r` is returned.
- This method overloads the function `pdivide` for polynomials.
- This method only exists if `Vars` consists of a single variable.

Method `pquo`: pseudo-quotient of polynomials

`pquo(`*`dom`* `a, `*`dom`* `b)`

- Computes the pseudo-quotient of `a` by `b`. For details see method `"pdivide"`.
- This method only exists if `Vars` consists of a single variable.

Method `prem`: pseudo-remainder of polynomials

`prem(`*`dom`* `a, `*`dom`* `b)`

- Computes the pseudo-remainder of `a` by `b`. For details see method `"pdivide"`.
- This method only exists if `Vars` consists of a single variable.

Method `random`: creates a random polynomial

```
random()
```

- Returns a random generated polynomial of this domain.
- With every call the global variable `SEED` is changed by a call of `random()`. Thus it is hard to create the same random sequence twice, see `random`.
- If the parameter `Vars` is the empty list, first a list of 1 to 4 variables is generated randomly and the random polynomial is generated in these indeterminates afterwards.
- This method overloads the function `polylib::randpoly` for polynomials.

Method `realSolve`: isolates all real roots of a real univariate polynomial

```
realSolve(dom a <, positive real number eps>)
```

- `realSolve(a)` returns intervals isolating the real roots of the real univariate polynomial `a`.
- `realSolve(a, eps)` returns refined intervals approximating the real roots of `a` to the relative precision given by `eps`.
- For a detailed description see function `polylib::realroots`.
- All coefficients must be convertible into either integers, rational numbers or (real) floating point numbers.
- This method overloads the function `polylib::realroots` for polynomials.
- This method only exists if `Vars` consists of a single variable.

Method `resultant`: resultant of two polynomials

```
resultant(dom a, dom b <, indeterminate var>)
```

- `resultant(a, b)` returns the resultant of `a` and `b` with respect to their main variable, i.e., the return value of the call `a::dom::mainvar(a)`.
- `resultant(a, b, var)` returns the resultant of `a` and `b` with respect to the variable `var`.
- The value returned is a polynomial of this domain or `FAIL`.
- This method overloads the function `polylib::resultant` for polynomials.
- This method only exists if `R` has the method `"_divide"`.

Method `ringmult`: multiplies a polynomial with a coefficient ring element

`ringmult(`*`dom`* `a, `*`R`* `c)`

- Multiplies `a` by the coefficient ring element `c`.

Method `solve`: zeros of polynomials

`solve(`*`dom`* `a` <, *`indeterminate`* `var`> <, `options`>`)`

`solve(`*`dom`* `a` <, *`list or set of indeterminates`* `vars`> <, `options`>`)`

`solve(`*`list or set of dom`* `ais` <, *`indeterminate`* `var`> <, `options`>`)`

`solve(`*`list or set of dom`* `ais` <, *`list or set of indeterminates`* `vars`> <, `options`>`)`

- `solve(a, ..)` tries to find the zeros of `a`. It is possible to control the behavior of `solve` by passing a variable or a set or list of variables together with options.
- `solve(ais, ..)` tries to find the zeros of the polynomial system `ais`. The exact behavior depends on further arguments.
- For a detailed description of possible return values and options see function `solve`.
- This method overloads the function `solve`.

Method `SPolynomial`: computes the S-polynomial of two polynomials

`SPolynomial(`*`dom`* `a, `*`dom`* `b` <, *`monomial ordering`* `ord`>`)`

- Computes the S-polynomial of `a` and `b` with respect to the monomial ordering `ord` (or w.r.t. `Order`, if `ord` is not explicitly given).
- This method is merely an interface for the function `groebner::spoly`.
- This method only exists if `R` is a field, i.e., a domain of category `Cat::Field`, and `Vars` is not the empty list.

Method `sqrfree`: square-free factorization of polynomials

`sqrfree(`*`dom`* `a)`

- Returns the square-free factorization of `a` as a list in the form `[u, a1, e1, ..., an, en]` which means that `a` = `u * a1^e1 * ... * an^en`.
- The `ai` are primitive and pairwise different square-free divisors of `a` and represented as elements of this domain. `u` is a unit of the coefficient ring and represented as an element of this domain. The `ei` are integers.

- This method overloads the function `polylib::sqrfree` for polynomials.
- This method only exists if `R` is a field, i.e., a domain of category `Cat::Field`, or if `R` is `Dom::Integer`.

Access Methods

Method `coeff`: coefficients of a polynomial

```
coeff(dom a)
coeff(dom a, indeterminate var, NonNegativeInteger n)
coeff(dom a, NonNegativeInteger n)
```

- `coeff(a)` returns a sequence with all coefficients of `a` as elements of the coefficient ring `R`. The coefficients are ordered according to the monomial ordering `Order`.
- `coeff(a,var,n)` returns the coefficient of the term `var^n`—as an element of this domain if it is of category `Cat::Polynomial(R)`, or as an element of the coefficient ring `R` if it is of `Cat::UnivariatePolynomial(R)`, where `a` is considered as a univariate polynomial in a valid variable `var`.
- `coeff(a,n)` returns the coefficient of the term `var^n`—as an element of this domain if it is of category `Cat::Polynomial(R)`, or as an element of the coefficient ring `R` if it is of `Cat::UnivariatePolynomial(R)`, where `a` is considered as a univariate polynomial in `var` and `var` is the main variable of `a`, i.e., the variable returned by `dom::mainvar(a)`.
- This method overloads the function `coeff` for polynomials.

Method `degree`: degree of a polynomial

```
degree(dom a)
degree(dom a, indeterminate var)
```

- `degree(a)` returns the total degree of `a`.
- `degree(a, var)` returns the degree of `a` with respect to `var`.
- The degree of the zero polynomial is defined as zero.
- This method overloads the function `degree` for polynomials.

Method `degreevec`: vector of exponents of the leading term of a polynomial

```
degreevec(dom a <, monomial ordering ord>)
```

- Returns the vector of exponents of the leading term of `a` with respect to the monomial ordering `ord` as a list. If `var1^e1 * ... * varm^em` is the leading term then the list `[e1, ..., em]` is returned. If `ord` is not explicitely given, the ordering `Order` will be used instead.
- The degree vector of the zero polynomial is defined as a list of zeros.
- This method overloads the function `degreevec` for polynomials.

Method `euclideanDegree`: Euclidean degree function

`euclideanDegree(`*`dom`* `a)`

- Returns the Euclidean degree of `a`, which is here simply defined as the degree of `a`.
- This method only exists if `Vars` consists of a single variable.

Method `ground`: ground term of a polynomial

`ground(`*`dom`* `a)`

- Returns the term of `a` in which no variable of `a` occurs.
- This method overloads the function `ground` for polynomials.

Method `has`: existence of an object in a polynomial

`has(`*`dom`* `a,` *`any`* `obj)`

- Tests if `a` contains the object `obj`. Only complete subexpressions and operators are found. If the second argument is a list or set then these elements are tested to see if at least one of it is a subexpression of `a`. See function `has` for more details.
- This method overloads the function `has`.

Method `indets`: indeterminates of a polynomial

`indets(<`*`dom`* `a>)`

- Returns a set of all indeterminates of `a`.
- In case `Vars` is not the empty list, `indets` can be called without argument.
- Since this domain allows expressions as indeterminates, the returned set may contain expressions, too.
- This method overloads the function `indets` for polynomials.

Method `lcoeff`: leading coefficient of a polynomial

`lcoeff(`*`dom`* `a)`

`lcoeff(`*`dom`* `a <,` *`list of indeterminates`* `vars> <,` *`monomial ordering`* `ord>)`

- `lcoeff(a)` returns the leading coefficient of `a` with respect to the monomial ordering `Order` as an element of the coefficient ring `R`.
- `lcoeff(a, ord)` returns the leading coefficient of `a` with respect to the monomial ordering `ord` as an element of the coefficient ring `R`.
- `lcoeff(a, vars <, ord>)` returns the leading coefficient of `a` with respect to the variable list `vars` and the monomial ordering `ord` as an element of this domain if it is of category `Cat::Polynomial(R)`, or as an element of the coefficient ring `R` if it is of `Cat::UnivariatePolynomial(R)`.
  - If `ord` is not explicitly given, the lexicographical order `LexOrder` will be used instead.
  - It tries to convert `a` into a polynomial in the specified list of indeterminates `vars` over the coefficient ring `R` and returns `FAIL` if this conversions fails.
- This method overloads the function `lcoeff` for polynomials.

Method `ldegree`: lowest degree of a polynomial

`ldegree(`*`dom`* `a)`

`ldegree(`*`dom`* `a,` *`indeterminate`* `x)`

- `ldegree(a)` returns the lowest total degree of the terms of `a`.
- `ldegree(a, x)` returns the lowest degree of the variable `x` in `a`.
- This method overloads the function `ldegree` for polynomials.

Method `lmonomial`: leading monomial of a polynomial

`lmonomial(`*`dom`* `a <,` *`monomial ordering`* `ord>)`

`lmonomial(`*`dom`* `a <,` *`list of indeterminates`* `vars> <,` *`monomial ordering`* `ord> <, Rem>)`

- `lmonomial(a <, ord>)` returns the leading monomial of `a` with respect to the monomial ordering `ord` as an element of this domain. If `ord` is not explicitly given, the ordering `Order` will be used instead.
- `lmonomial(a, vars <, ord>)` returns the leading monomial of `a` with respect to the variable list `vars` and the monomial ordering `ord` as an element of this domain.

- If `ord` is not explicitly given, the lexicographical order `LexOrder` will be used instead.
- It tries to convert `a` into a polynomial in the specified list of indeterminates `vars` over the coefficient ring `R` and returns `FAIL` if this conversions fails.

- `lmonomial(a <, vars> <, ord>, Rem)` returns the list consisting of the leading monomial and the reductum of `a` with respect to the variable list `vars` and the monomial ordering `ord` as a list of elements of this domain.
  - If `ord` is not explicitly given, the lexicographical order `LexOrder` will be used instead.
  - It tries to convert `a` into a polynomial in the specified list of indeterminates `vars` over the coefficient ring `R` and returns `FAIL` if this conversions fails.
- In MuPAD a monomial denotes a coefficient together with a power product as, e.g., $3x^2$.

- This method overloads the function `lmonomial` for polynomials.

Method `lterm`: leading term of a polynomial

`lterm(`*dom* `a)`

`lterm(`*dom* `a <,` *list of indeterminates* `vars> <,` *monomial ordering* `ord>)`

- `lterm(a)` returns the leading term of `a` with respect to the monomial ordering `Order` as an element of this domain.
- `lterm(a, ord)` returns the leading coefficient of `a` with respect to the monomial ordering `ord` as an element of this domain.
- `lterm(a, vars <, ord>)` returns the leading term of `a` with respect to the variable list `vars` and the monomial ordering `ord` as an element of this domain.
  - If `ord` is not explicitly given, the lexicographical order `LexOrder` will be used instead.
  - It tries to convert `a` into a polynomial in the specified list of indeterminates `vars` over the coefficient ring `R` and returns `FAIL` if this conversions fails.
- In MuPAD a term denotes a power product without a coefficient as, e.g., x^2y^3z.

- This method overloads the function `lterm` for polynomials.

Method `mainvar`: main variable of a polynomial

`mainvar(<`*`dom`* `a>)`

- Returns the main indeterminate of `a`, i.e., the first element of `dom::orderedVariableList`
- If `Vars` is not the empty list, `mainvar` can be called without argument.

Method `mapcoeffs`: applies a function to the coefficients of a polynomial

`mapcoeffs(`*`dom`* `a,` *`function`* `f <,` *`sequence of arguments`* `e1, ... >)`

- Applies the function `f` to the coefficients of `a` and returns `a` with the new coefficients.
  - `mapcoeffs(a, f)`: `f(ci)` is executed for each coefficient `ci` of `a`.
  - `mapcoeffs(a, f, e1, ..., en)`: `f(ci, e1, ..., en)` is executed for each coefficient `ci` of `a` with the additional arguments `e1, ..., en`.
- This method overloads the function `mapcoeffs` for polynomials.

Method `multcoeffs`: multiplies the coefficients of a polynomial with a factor

`multcoeffs(`*`dom`* `a,` *`R`* `c)`

- Multiplies all the coefficients of `a` with the factor `c`.
- This method overloads the function `multcoeffs` for polynomials.

Method `nterms`: number of terms of a polynomial

`nterms(`*`dom`* `a)`

- Returns the number of non-zero terms of `a`. The zero polynomial has no terms.
- This method overloads the function `nterms` for polynomials.

Method `nthcoeff`: n-th coefficient of a polynomial

`nthcoeff(`*`dom`* `a,` *`integer`* `n <,`*`monomial ordering`* `ord>)`

- Returns the `n`-th (non-zero) coefficient of `a` with respect to the monomial ordering `ord`. If `ord` is not explicitly given, the ordering `Order` will be used instead.
- If `n` is larger than the number of monomials of the polynomial then the function returns `FAIL`.

- The zero polynomial has no monomials. `nthcoeff` returns `FAIL` when invoked on the zero polynomial.
- This method overloads the function `nthcoeff` for polynomials.

Method `nthmonomial`: n-th monomial of a polynomial

`nthmonomial(`*`dom`* `a,` *`integer`* `n <,`*`monomial ordering`* `ord>)`

- Returns the `n`-th (non-zero) monomial of `a` with respect to the monomial ordering `ord`. If `ord` is not explicitly given, the ordering `Order` will be used instead.
- If `n` is larger than the number of monomials of the polynomial then the function returns `FAIL`.
- The zero polynomial has no monomials. `nthmonomial` returns `FAIL` for the zero polynomial.
- This method overloads the function `nthmonomial` for polynomials.

Method `nthterm`: n-th term of a polynomial

`nthterm(`*`dom`* `a,` *`integer`* `n <,`*`monomial ordering`* `ord>)`

- Returns the `n`-th (non-zero) term of `a` with respect to the monomial ordering `ord`. If `ord` is not explicitly given, the ordering `Order` will be used instead.
- If `n` is larger than the number of monomials of the polynomial then the function returns `FAIL`.
- The zero polynomial has no monomials. `nthterm` returns `FAIL` when called with the zero polynomial.
- This method overloads the function `nthterm` for polynomials.

Method `orderedVariableList`: ordered list of indeterminates of a polynomial

`orderedVariableList(<`*`dom`* `a>)`

- Returns the ordered list of variables as specified when creating the domain. If no variable was specified for this domain (i.e., `Vars = []`, a sorted list of all variables of `a` is returned.
- In case `Vars` is not the empty list, `orderedVariableList` can be called without an argument.

Method `pivotSize`: size of a pivot element

```
pivotSize(dom a)
```

- Returns the "size" of `a` as a pivot element, which is simply the total degree of `a`.
- This method is called if this domain is used as the component ring of a matrix domain to perform a Gaussian elimination.

Method `reductum`: reductum of a polynomial

```
reductum(dom a <,monomial ordering ord>)
```

- Returns the reductum of `a` (i.e., `a-lmonomial(a)`) with respect to the monomial ordering `ord`. If `ord` is not explicitly given, the ordering `Order` will be used instead.

Method `tcoeff`: lowest coefficient of a polynomial

```
tcoeff(dom a <,monomial ordering ord>)
```

- Returns the lowest coefficient of `a` with respect to the monomial ordering `ord`. If `ord` is not explicitly given, the ordering `Order` will be used instead.
- This method overloads the function `tcoeff` for polynomials.

Conversion Methods

Method `convert`: conversion to a polynomial

```
convert(any p)
```

- Tries to convert a polynomial expression or a polynomial `p` to an element of this domain and returns either an element of this domain or `FAIL`.

Method `expr`: conversion to a basic type

```
expr(dom a)
```

- Converts the polynomial `a` of this domain into an element of a basic domain. Thus the coefficients may no longer belong to the coefficient ring `R`.
- This method overloads the function `expr`.

Method `poly`: converts to a basic polynomial domain

```
poly(dom a)
```

- Converts the polynomial `a` of this domain into an element of the basic domain `DOM_POLY`.
- This method overloads the function `poly`.

Method `TeX`: TeX formatting of a polynomial

```
TeX(dom a)
```

- Returns a TEX-formatted string for the polynomial `a`.

Method `TeXCoeff`: TeX formatting of a polynomial coefficient

```
TeXCoeff(R c)
```

- Returns a TEX-formatted string for the polynomial coefficient `c`.

Method `TeXident`: TeX formatting of a polynomial indeterminate

```
TeXident(indeterminate var)
```

- Returns a TEX-formatted string for the polynomial indeterminate `var`.

Method `TeXTerm`: TeX formatting of a polynomial term

```
TeXTerm(dom t)
```

- Returns a TEX-formatted string for the polynomial term `t`.

Technical Methods

Method `adaptIndets`: converts polynomials to common indeterminates

```
adaptIndets(<dom a, dom b , ...>)
```

- Computes a common sorted list of indeterminates of all arguments and returns the sequence of polynomials converted to polynomials of type `DOM_POLY` with respect to this list of indeterminates.
- This method only exists if the parameter `Vars` is the empty list (`[]`).

Method `isNeg`: test on leading output token

`isNeg(`*`dom`* `a)`

- Tests the leading output token of `a` and returns `TRUE` if it is the minus token `-` and `FALSE` if that is not the case or MuPAD cannot determine that token (e.g., if a domain involved does not have this method).

Method `mult`: multiplies polynomials

`mult(`*`dom`* `a,` *`dom`* `b, ...)`

- Multiplies an arbitrary number of elements of this domain.

Method `new`: creates a new element

`new(`*`any`* `p)`

`new(`*`list of lists`* `lm)`

`new(`*`list of lists`* `lm,` *`list of indeterminates`* `v)`

- One may use this method either in the form `dom::new(p)` or in the form `dom(p)`.
- `dom(p)` creates an element of this domain from a polynomial or a polynomial expression `p` and returns that element. If this is not possible, an error message is given.
- If `Vars` is chosen as the empty list (`[]`) then in creating new elements from a polynomial or polynomial expression the function `indets` is first called to get the identifiers. Afterwards the element is created with this list of identifiers. For creating an element from a constant the dummy variable `_dummy` is introduced. The drawback of this approach is that two mathematically equal polynomials may have variable lists which differ by the dummy variable.
- `dom(lm)` creates, if `Vars` $\neq$ `[]`, a polynomial from the list `lm` of the form `[[c1, [e11,... e1n]],... [cm,[em1,... emn]]]` where the `ci` are coefficients and the `eij` are the exponents with respect to `Vars`. For a univariate polynomial this list can be simplified to `[[c1,e1],... [cm,em]]`.
- `dom(lm,v)` creates, if `Vars` $=$ `[]`, a polynomial from the list `lm` of the form `[[c1, [e11,... e1n]],... [cm,[em1,... emn]]]` where the `ci` are coefficients and the `eij` are the exponents with respect to `v`. For a univariate polynomial this list can be simplified to `[[c1,e1],... [cm,em]]`. The list of indeterminates `v` must contain valid indeterminates.

Method `plus`: adds polynomials

`plus(`*`dom`* `a, `*`dom`* `b, ...)`

- Adds an arbitrary number of elements of this domain.

Method `print`: prints polynomials

`print(`*`dom`* `a)`

- Prints the polynomial `a`. The terms are printed with respect to the given monomial ordering `Order` in descending order.
- This method overloads the function `print`.

Method `printMonomial`: prints a monomial in defined order

`printMonomial(`*`R`* `c, `*`list of NonNegativeIntegers`* `d, `*`list of indeterminates`* `v)`

- Returns an ordered expression `c*v[1]^d[1]*..`, where `d` is the degree vector with respect to variable list `v`.

Method `printTerm`: prints a term in defined order

`printTerm(`*`list of NonNegativeIntegers`* `d)`

`printTerm(`*`list of NonNegativeIntegers`* `d, `*`list of indeterminates`* `v)`

- `printTerm(d)` returns an ordered sequence of the indeterminates together with their powers as given in `Vars` and the degree vector `d` respectively.

  Note that this call is only valid if `Vars` is not the empty list.
- `printTerm(d,v)` returns an ordered sequence of the indeterminates together with their powers as given in the variable list `v` and the degree vector `d` respectively.

  Note that this call is only valid if `nops(v)=nops(d)`.

Method `Rep`: data representation of a polynomial

`Rep(`*`dom`* `a)`

- Returns the internal data representation of `a`.

Method `sign`: leading sign of a polynomial

`sign(`*dom* `a)`

- Determines the sign of `a`, which is 0 if `a` is zero, 1 if `a` is either a positive constant or a positive monomial, `sign(a)` if `nterms(a)` > 1 and -1 otherwise. This method is currently used within the `"printMonomial"` method for pretty printing elements of this domain and is more or less intended to be an internal procedure. It is planned to replace this method by the method `"isNeg"` in future versions.
- **Note:** this method does not have the meaning of a mathematical sign function!

Example 1. The following call creates a polynomial domain in x, y and z.

```
>> DP := Dom::DistributedPolynomial([x, y, z])

 Dom::DistributedPolynomial([x, y, z],

    Dom::ExpressionField(normal, iszero@normal), LexOrder)
```

Since neither the coefficient ring nor the monomial ordering was specified, this domain is created with the default values for these parameters.

It is rather easy to create elements of this domain, as e.g.

```
>> a := DP(x + 2*y*z + 3)

                          x + 2 y z + 3

>> b := DP(z^4 - 2*y^2*x^2)

                              2  2    4
                         - 2 x  y  + z
```

In contrast to expressions all elements of this domain have a representation which is fixed by the chosen `Order`, the representation of the coefficient ring `R` and the way of representing monomials.

With these elements one can now perform usual arithmetic operations as, e.g., (scalar) multiplication, multiplication with integers and adding polynomials and ring elements:

```
>> 4*b^2 + a/3 + 1/2

          4  4       2  2  4                          8
      16 x  y  - 16 x  y  z  + 1/3 x + 2/3 y z + 4 z  + 3/2
```

There are a lot of methods for manipulating polynomials and to get access to all parts of a polynomial. For example one has access to the leading monomial of `a` as follows:

```
>> lmonomial(a)
```

```
                                    x
```

The leading monomial of a polynomial depends on the monomial ordering, so with respect to the degree order one gets a different result:

```
>> lmonomial(a, DegreeOrder)
```

```
                                  2 y z
```

To get `a` minus its leading monomial one may call:

```
>> DP::reductum(a)
```

```
                               2 y z + 3
```

Obviously the following identity holds:

```
>> a - lmonomial(a) - DP::reductum(a)
```

```
                                    0
```

There are also methods for converting elements of this domain into other domains, like a basic polynomial domain or the domain of arbitrary expressions:

```
>> poly(a), domtype(poly(a))
```

```
 poly(x + 2 y z + 3, [x, y, z], Dom::ExpressionField(normal,

    iszero@normal)), DOM_POLY
```

```
>> expr(b), domtype(expr(b))
```

```
                      4      2  2
                     z  - 2 x  y , DOM_EXPR
```

Super-Domain: `Dom::BaseDomain`

Axioms

- **if `R` has `Ax::normalRep`, then**
  `Ax::normalRep`
- **if `R` has `Ax::canonicalRep`, then**
  `Ax::canonicalRep`

Changes:

- `Dom::DistributedPolynomial` used to be `Dom::PolynomialExplicit`.
- The former implementation of this domain does no longer exist and is replaced by the former (undocumented) domain constructor `Dom::PolynomialExplicit`.
- It is now allowed to call this domain with zero, one, two or three arguments. With the third argument one can now choose an appropriate monomial ordering.
- The method `"indets"` now returns a set of indeterminates.
- The methods `"Rep"`, `"SPolynomial"`, `"decompose"`, `"dimension"`, `"func_call"`, `"groebner"`, `"ground"`, `"int"`, `"makeIntegral"`, `"monic"`, `"numericSolve"`, `"orderedVariableList"`, `"ordering"`, `"realSolve"`, `"reductum"`, `"resultant"`, `"variables"` were added.
- The method `"Factor"` was removed.

`Dom::Expression` – the domain of all MuPAD objects of basic type

`Dom::Expression` comprises all objects only consisting of operands of built-in types.

Creating Elements:

- `Dom::Expression(x)`

Parameters:

`x` — An object of basic type consisting only of operands of built-in types, or any other object convertible to such using `expr`.

Categories:

`Cat::BaseCategory`

Related Domains: `Dom::ExpressionField`

Details:

- `Dom::Expression` is a façade domain: it has no domain elements, but uses system representation.
- Unlike `Dom::ExpressionField`, `Dom::Expression` does not belong to any arithmetical category, and its elements need not be arithmetical expressions.
- `Dom::Expression` mainly serves as a super–domain to `Dom::ArithmeticalExpression`; it rarely makes sense to use it directly.

Entries:

`randomIdent` an identifier used for creating random elements

Conversion Methods

Method `convert`: conversion of objects

```
convert(any x)
```

- `expr` is used to convert `x` into an object of basic type. This method does *not* create elements of `Dom::Expression`, since `Dom::Expression` has system representation.

Method `convert_to`: conversion to other domains

```
convert_to(expression x, any T)
```

- This method always returns `FAIL` unless `T` is `Dom::Expression`; in the latter case, `x` is returned.

Method `expr`: just return the argument

```
expr(expression x)
```

- Since the argument(s) are expressions already, they are just returned.

Method `testtype`: tests whether its argument is an expression

```
testtype(any x, Dom::Expression)
```

- This method returns `TRUE` if `x` can be converted to an expression, and `FAIL` otherwise. The second argument must equal `Dom::Expression`; this is not checked.
- This method overloads `testtype`; since `Dom::Expression` has no domain elements, the overloading can only be caused by the second argument.

Method `float`: convert numbers to floats

```
float(expression x)
```

- This method is identical to the kernel function `float`.

Technical Methods

Method `subs`: substitution

`subs(`*`expression`* `x,` *`substitution`* `s, ...)`

- This is just the method `subs` of the standard library; see there for details about the calling syntax.

Method `subsex`: extended substitution

`subsex(`*`expression`* `x,` *`substitution`* `s, ...)`

- This is just the method `subsex` of the standard library; see there for details about the calling syntax.

Method `random`: create random expression

`random()`

- This method returns a randomly chosen rational expression in one variable.

Example 1. Almost every MuPAD object can be converted to an expression. Objects of basic type *are* expressions.

```
>> Dom::Expression([3, array(1..2), rectform(exp(I))])

           --    +-          -+                      --
           |  3, | ?[1], ?[2] |, cos(1) + I sin(1)  |
           --    +-          -+                      --
```

The `convert` method flattens its argument: hence expression sequences are *not* allowed.

```
>> Dom::Expression((3, x))

 Error: expecting one argument [Dom::Expression::convert]
```

Super-Domain: `Dom::BaseDomain`

Axioms

`Ax::systemRep`, `Ax::efficientOperation("_divide")`,
`Ax::efficientOperation("_mult")`,
`Ax::efficientOperation("_invert")`

Changes:

- No changes.

`Dom::ExpressionField` – the domains of expressions forming a field

`Dom::ExpressionField(Normal, IsZero)` creates a domain of expressions forming a field, where the functions `Normal` and `IsZero` are used to normalize expressions and test for zero.

Domain:

- `Dom::ExpressionField(<Normal <, IsZero>>)`

Parameters:

`Normal` — a function used to normalize the expressions of the domain; default is `id`.

`IsZero` — a function used to test the expressions of the domain for zero; default is `iszero @ Normal`.

Details:

- `Dom::ExpressionField(Normal, IsZero)` creates a domain which is supposed to be a field, where the field elements are represented as expressions. The function `Normal` is used to normalize the expressions representing the elements, the function `IsZero` is used to test the expressions for zero. It is assumed that the field has characteristic 0.
- The domain cannot decide if the element expressions—given the normalizing function and zero test—actually form a field. It is up to the user to choose correct functions for normalizing and zero test and to enter only valid expressions as domains elements.
- One should view this domain constructor as a pragmatic way to create a field of characteristic 0 in an ad-hoc fashion. Note that the default of using `id` and `iszero` does not yield a field really, but it is often convenient and sensible to use the resulting structure as a field.
- `Normal` must be a function which takes an expression representing a domain element and returns the normalized expression. `Normal` should return `FAIL` if the expression is not valid.
- If `Normal` is not given, then the system function `id` is used, i.e., only the kernel simplifier is used to normalize expressions.

- If a normalizing function other than `id` is given, it is assumed that this functions returns a normal form where the zero element is uniquely represented by the constant `0`.
- `IsZero` must be a function which takes an expression representing a domain element and returns `TRUE` if the expression represents zero and `FALSE` otherwise.
- If `IsZero` is not given, then `iszero @ Normal` is used for zero testing. If `Normal` is equal to `id` this functional expression is simplified to `iszero`.
- If `Normal` is equal to `id` and `IsZero` is equal to `iszero`, a façade domain is created, i.e., the domain elements are simply expressions and are not explicitly created by `new`.

  Otherwise the elements of the domain are explicitly created by `new`. Each such element has one operand, which is the expression representing the domain element. The element expressions are normalized after each operation using the function `Normal`.

Categories:

`Cat::Field`, `Cat::DifferentialRing`

`Dom::ExpressionField(Normal, IsZero)(e)` creates a field element represented by the expression `e`.

Creating Elements:

- `Dom::ExpressionField(Normal, IsZero)(e)`

Parameters:

`e` — an expression representing a field element.

Details:

- `Dom::ExpressionField(Normal, IsZero)(e)` creates a field element represented by the expression `e`. The expression is normalized using the function `Normal`.
- If `Normal` returns `FAIL`, it is assumed that the expression does not represent a valid field element. If this test is not fully implemented the domain cannot decide if the expression represents a valid field element. In this case it is up to the user to enter only valid expressions as field elements.

- If `Normal` is equal to `id` and `IsZero` is equal to `iszero`, the domain is only a façade domain. In this case the expression `e` is returned after being simplified by the built-in kernel simplifier.

Entries:

`characteristic` The characteristic of the fields created by this constructor is assumed to be `0`.

`one` The element represented by the expression `1` is assumed to be a neutral element w.r.t. `"_mult"`.

`zero` The element represented by the expression `0` is assumed to be a neutral element w.r.t. `"_plus"`.

Mathematical Methods

Method `abs`: absolute value

`abs(`*`dom`* `x)`

- Returns the absolute value of `x`. Maps `abs` to the expression representing `x`. See `abs` for details.
- Overloads the function `abs`, thus may be called via `abs(x)`.

Method `combine`: combines terms of the same algebraic structure

`combine(`*`dom`* `x <, a>)`

- Combines terms of the same algebraic structure. Maps `combine` to the expression representing `x`. See `combine` for details and optional additional arguments.
- Overloads the function `combine`, thus may be called via `combine(x,...)`.

Method `conjugate`: complex conjugate

`conjugate(`*`dom`* `x)`

- Returns the complex conjugate of `x`. Maps `conjugate` to the expression representing `x`. See `conjugate` for details.
- Overloads the function `conjugate`, thus may be called via `conjugate(x)`.

Method `D`: differential operator

`D(<list l,> dom x)`

- Returns the derivative of `x`, where `x` is viewed as functional expression. Maps `D` to the expression representing `x`. See `D` for details and a description of the optional additional argument `l`.
- Overloads the function `D`, thus may be called via `D(x)` or `D(l, x)`.

Method `denom`: denominator

`denom(dom x)`

- Returns the denominator of `x`. Maps `denom` to the expression representing `x`. See `denom` for details.
- Overloads the function `denom`, thus may be called via `denom(x)`.

Method `diff`: differentiates an element

`diff(dom x <, v ...>)`

- Differentiates `x` with respect to the remaining arguments. Maps `diff` to the expression representing `x`. See `diff` for details and optional additional arguments.
- Overloads the function `diff`, thus may be called via `diff(x,...)`.

Method `_divide`: divides elements

`_divide(dom x, dom y)`

- Computes `x/y` by dividing the expressions representing `x` and `y`.
- Overloads the function `_divide`, thus may be called via `x/y` or `_divide(x, y)`.

Method `equal`: test for mathematical equality

`equal(dom x, dom y)`

- Tests if `x` is mathematically equal to `y`. This is implemented by testing if `x-y` is zero.

Method `expand`: expands an element

`expand(dom x)`

- Expands `x` by mapping `expand` to the expression representing `x`.
- Overloads the function `expand`, thus may be called via `expand(x)`.

Method `factor`: factorizes an element

`factor(`*`dom`* `x)`

- Returns the factorization of `x` by mapping `factor` to the expression representing `x`.
- Overloads the function `factor`, thus may be called via `factor(x)`.

Method `float`: floating-point approximation

`float(`*`dom`* `x)`

- Returns a floating-point approximation of `x` by mapping `float` to the expression representing `x`.
- Overloads the function `float`, thus may be called via `float(x)`.

Method `gcd`: greatest common divisor

`gcd(`*`dom`* `x, ...)`

- Computes a greatest common divisor of the arguments by mapping the function `gcd` to the expressions representing the arguments.
- Overloads the function `gcd`, thus may be called via `gcd(x, ...)`.

Method `Im`: imaginary part of an element

`Im(`*`dom`* `x)`

- Returns the imaginary part of `x`. Maps `Im` to the expression representing `x`. See `Im` for details.
- Overloads the function `Im`, thus may be called via `Im(x)`.

Method `int`: definite and indefinite integration

`int(`*`dom`* `x <, v>)`

- Computes the definite or indefinite formal integral of `x` by mapping `int` to the expression representing `x`. See `int` for details and additional arguments.
- Overloads the function `int`, thus may be called via `int(x,...)`.

Method `intmult`: integer mutiple

`intmult(`*`dom`* `x,` *`DOM_INT`* `n)`

- Returns the integer multiple `x*n` by multiplying the expression representing `x` by `n`.

Method `_invert`: inverts an element

`_invert(`*`dom`* `x)`

- Returns the inverse `1/x` of `x` by computing the inverse of the expression representing `x`.
- Overloads the function `_invert`, thus may be called via `1/x` or `_invert(x)`.

Method `iszero`: test for zero

`iszero(`*`dom`* `x)`

- Tests if `x` is zero by calling `IsZero` with the expression representing `x` as argument.
- Overloads the function `iszero`, thus may be called via `iszero(x)`.

Method `lcm`: least common multiple

`lcm(`*`dom`* `x, ...)`

- Computes a least common multiple of the arguments by mapping the function `lcm` to the expressions representing the arguments.
- Overloads the function `lcm`, thus may be called via `lcm(x, ...)`.

Method `_leequal`: tests if less or equal

`_leequal(`*`dom`* `x,` *`dom`* `y)`

- Tests if `x` is less than or equal to `y` by mapping the function `_leequal` to the arguments.
- Please note that the function `_leequal` can only test numbers (in a syntactical sense), but not constant expressions like `PI` or `sqrt(2)`.
- Overloads the function `_leequal`, thus may be called via `x <= y`, `y >= x` or `_leequal(x, y)`.

Method `_less`: tests if element is less

`_less(`*`dom`* `x,` *`dom`* `y)`

- Tests if `x` is less than `y` by mapping the function `_less` to the arguments.
- Please note that the function `_less` can only test numbers (in a syntactical sense), but not constant expressions like `PI` or `sqrt(2)`.
- Overloads the function `_less`, thus may be called via `x < y`, `y > x` or `_less(x, y)`.

Method `limit`: limit computation

`limit(`*dom* `x <, v, ...>)`

- Computes the limit of `x` by mapping the function `limit` to the expression representing `x`. See `limit` for details and additional arguments.
- Overloads the function `limit`, thus may be called via `limit(x,...)`.

Method `max`: maximum of arguments

`max(`*dom* `x, ...)`

- Computes the maximum of the arguments by mapping the function `max` to the expressions representing the arguments.
- Overloads the function `max`, thus may be called via `max(x,...)`.

Method `min`: minimum of arguments

`min(`*dom* `x, ...)`

- Computes the minimum of the arguments by mapping the function `min` to the expressions representing the arguments.
- Overloads the function `min`, thus may be called via `min(x,...)`.

Method `_mult`: multiplies elements

`_mult(`*dom* `x, ...)`

- Returns the product of the arguments.
- If all arguments are of this domain or can be coerced to this domain (using the method `coerce`), the product of the expressions representing the arguments is calculated using the function `_mult`.
  If one of the arguments cannot be coerced, the arguments up to the offending one are multiplied and then the method `"_mult"` of the domain of the offending argument is called to multiply the remaining arguments.
- Overloads the function `_mult`, thus may be called via `x*...` or `_mult(x,...)`.

Method `_negate`: negates an element

`_negate(`*dom* `x)`

- Returns the negative `-x` of `x` by computing the negative of the expression representing `x`.
- Overloads the function `_negate`, thus may be called via `-x` or `_negate(x)`.

Method `norm`: norm of an element

```
norm(dom x)
```

- Computes the norm of `x` as the absolute value of the expression representing `x`. See the function `abs` for details.
- Overloads the function `norm`, thus may be called via `norm(x)`.
- Please note that the system function `norm`, applied to an expression, computes the norm of that expression interpreted as a polynomial expression and *not* the absolute value of the expression. This may be regarded as an inconsistency.

Method `normal`: normal form

```
normal(dom x)
```

- Computes the normal form of `x` by applying the function `Normal` to the expression representing `x`.
- Overloads the function `normal`, thus may be called via `normal(x)`.

Method `numer`: numerator

```
numer(dom x)
```

- Returns the numerator of `x`. Maps `numer` to the expression representing `x`. See `numer` for details.
- Overloads the function `numer`, thus may be called via `numer(x)`.

Method `_plus`: adds elements

```
_plus("dom" x, ...)
```

- Returns the sum of the arguments.
- If all arguments are of this domain or can be coerced to this domain (using the method `coerce`) the sum of the expressions representing the arguments is calculated using the function `_plus`.

  If one of the arguments cannot be coerced the arguments up to the offending one are added and then the method `"_plus"` of the domain of the offending argument is called to add the remaining arguments.
- Overloads the function `_plus`, thus may be called via `x+...` or `_plus(x,...)`.

Method `_power`: exponentiates arguments

```
_power(dom x, any y)
_power(any x, dom y)
```

- Returns `x` to the power of `y`.
- If both arguments are of this domain the power is calculated by mapping the function `_power` to the expressions representing the arguments.

  If one of the arguments is not of this domain it is coerced to this domain, then the power is computed. If the coercion fails an error is raised.

  Note that it is assumed that at least one of the arguments is of this domain.
- Overloads the function `_power`, thus may be called via `x^y` or `_power(x, y)`.

Method `radsimp`: simplifies radicals

```
radsimp(dom x)
```

- Simplifies radicals in `x`. Maps the function `radsimp` to the expression representing `x`. See `radsimp` for details.
- Overloads the function `radsimp`, thus may be called via `radsimp(x)`.

Method `random`: creates a random element

```
random()
```

- Creates a random element of this domain by creating a univariate random polynomial expression which is then normalized using the function `Normal`.
- See `polylib::randpoly` for details about creating random polynomials.

Method `Re`: real part of an element

```
Re(dom x)
```

- Returns the real part of `x`. Maps `Re` to the expression representing `x`. See `Re` for details.
- Overloads the function `Re`, thus may be called via `Re(x)`.

Method `sign`: sign of an element

`sign(`*`dom`* `x)`

- Computes the sign of `x` by mapping the function `sign` to the expression representing `x`. See `sign` for details.
- Overloads the function `sign`, thus may be called via `sign(x)`.

Method `simplify`: general simplification of an element

`simplify(`*`dom`* `x <, a>)`

- Tries to simplify `x` by mapping the function `simplify` to the expression representing `x`. See `simplify` for details and optional additional arguments.
- Overloads the function `simplify`, thus may be called via `simplify(x,...)`.

Method `solve`: solves an equation

`solve(`*`dom`* `x <, a, ...>)`

- Tries to solve the equation `x = 0` using the standard solver function `solve`. Maps `solve` to the expression representing `x` and the additional arguments and directly returns the result of `solve`.
- Note that this method will never return an element of this domain. See `solve` for details about results and optional additional arguments.
- Overloads the function `solve`, thus may be called via `solve(x,...)`.

Method `sqrfree`: square-free factorization

`sqrfree(`*`dom`* `x)`

- Returns the square-free factorization of `x` by mapping `polylib::sqrfree` to the expression representing `x`.
- Overloads the function `polylib::sqrfree`, thus may be called via `polylib::sqrfree(x)`.

Method `_subtract`: subtracts elements

`_subtract(`*`dom`* `x, `*`dom`* `y)`

- Computes `x-y` by subtracting the expressions representing `x` and `y`.
- Overloads the function `_subtract`, thus may be called via `x-y` or `_subtract(x, y)`.

Conversion Methods

Method `convert`: convert to this domain

`convert(`*`any`* `x)`

- Tries to convert `x` to an element of this domain:
  - If `x` is from a domain defined by `Dom::ExpressionField` the expression representing `x` is used as the expression representing the new element of this domain.
  - Otherwise `x` is converted to an expression using the function `expr` and the resulting expression is used to represent the new element.

  Returns `FAIL` if the conversion fails.

Method `convert_to`: convert to other domain

`convert_to(`*`dom`* `x,` *`DOM_DOMAIN`* `T)`

- Tries to convert `x` to an element of the domain `T`:
  - If `T` is defined by `Dom::ExpressionField` the expression representing `x` is used as the expression representing the new element of `T`.
  - Otherwise the method `"convert"` of `T` is called with the expression representing `x` as argument.

  Returns `FAIL` if the conversion fails.

Method `expr`: convert to basic type

`expr(`*`dom`* `x)`

- Converts `x` to an expression containing only elements of basic types. Maps `expr` to the expression representing `x`.
- This method is called by the function `expr` if a subexpression of the argument is an element of this domain.

Method `new`: creating an element

`new(`*`any`* `x)`

- Tries to create an element of this domain given `x`: First the method `"convert"` of this domain and then, if this fails, the method `"convert_to"` of the domain of `x` is called. If both methods fail an error is raised.
- Overloads the function call operator for this domain, thus may be called via `F(x)` where `F` is this domain.

Access Methods

Method `nops`: number of operands

`nops(`*`dom`* `x)`

- Returns the number of operands of the expression representing `x`. See `nops` for details.
- Overloads the function `nops`, thus may be called via `nops(x)`.

Method `op`: get operands

`op(`*`dom`* `x)`

- Returns an expression sequence with the operands of the expression representing `x`. The operands are converted to elements of this domain.

`op(`*`dom`* `x,` *`NonNegInt`* `i)`

- Returns the operand with index `i` of the expression representing `x`. If `i` is `0` then the operator of the expression is returned, which usually is not an element of this domain. The other operands are converted to elements of this domain.
- This method is called by the function `op` when an element of this domain is contained, as a subexpression, in the first argument of `op`. Operand ranges and paths are handled by `op` and need not be handled by this method. See `op` for details.

Method `subs`: substitute subexpressions

`subs(`*`dom`* `x,` *`equation`* `e, ...)`

- Substitutes complete subexpressions in the expression representing `x` as specified by the substitution equations `e, ...`. The equations must be of the form `o = n`, where `o` is the original subexpression to replace and `n` is the new value that is inserted instead.
- Maps `subs` to the expression representing `x`. The resulting expression is converted to an element of this domain.
- This method is called by the function `subs` when an element of this domain is contained, as a subexpression, in the first argument of `subs`. See `subs` for details.

Method `subsex`: extended substitution

`subsex(`*`dom`* `x, `*`equation`* `e, ...)`

- Substitutes partial subexpressions in the expression representing `x` as specified by the substitution equations `e, ...`. The equations must be of the form `o = n`, where `o` is the original partial subexpression to replace and `n` is the new value that is inserted instead.
- Maps `subsex` to the expression representing `x`. The resulting expression is converted to an element of this domain.
- This method is called by the function `subsex` when an element of this domain is contained, as a subexpression, in the first argument of `subsex`. See `subsex` for details.

Method `subsop`: substitute operand

`subsop(`*`dom`* `x, `*`equation`* `e)`

- Substitutes in the expression representing `x` the operand given by the substitution equation `e`. The equation must be of the form `i = v`, where `i` is the index of the operand and `v` is its new value.

  Maps `subsop` to the expression representing `x`. The resulting expression is converted to an element of this domain.
- This method is called by the function `subsop` when an element of this domain is contained, as a subexpression, in the first argument of `subsop`. Operand ranges and pathes are handled by `subsop` and need not be handled by this method. See `subsop` for details.

Technical Methods

Method `indets`: the identifiers of an element

`indets(`*`dom`* `x <, `*`option`* `optionName>)`

- Returns a set of the identifiers contained in `x`. An identifier is an object of the domain `DOM_IDENT`. See `indets` for details and available options.
- Overloads the function `indets`, thus may be called via `indets(x)` and `indets(x, optionName)`, respectively.

Method `length`: size of an element

`length(`*`dom`* `x)`

- Returns the size of `x`. Maps `length` to the expression representing `x`. See `length` for details.
- Overloads the function `length`, thus may be called via `length(x)`.

Method `map`: applies function to operands

`map(`*`dom`* `x, `*`function`* `f <, a, ...>)`

- Applies the function `f` to the operands of the expression representing `x`. Additional arguments `a, ...` may be handled to the function. See `map` for details.
- Overloads the function `map`, thus may be called via `map(x, f,...)`.

Method `rationalize`: approximate floating point numbers by rationals

`rationalize(`*`dom`* `x <, a, ...>)`

- Replaces all floating point numbers in the expression representing `x` by rational numbers. Maps the function `numeric::rationalize` to the expression representing `x`. See `numeric::rationalize` for details and additional optional arguments.
- Note that this method does *not* overload the function `rationalize` from the standard library package, but the function `numeric::rationalize` from the `numeric` package instead. Thus the method may be called via `numeric::rationalize(x,...)`.

Method `pivotSize`: pivot size

`pivotSize(`*`dom`* `x)`

- Returns the pivot size used during Gaussian elemination, which is computed as `length` of the expression representing `x`.

Super-Domain:

if `Normal` = `id` and `IsZero` = `iszero` then
`Dom::ArithmeticalExpression`

else
`Dom::BaseDomain`

Axioms

if `Normal` = `id` then
`Ax::efficientOperation("_divide"),`
`Ax::efficientOperation("_mult"),`
`Ax::efficientOperation("_invert"),`
if `IsZero` = `iszero` then
`Ax::systemRep`

else
`Ax::normalRep`

Example 1. `Dom::ExpressionField(normal)` creates a field of rational expressions over the rationals. The expressions representing the field elements are allways normalized by `normal`:

```
>> Fn := Dom::ExpressionField(normal):
   a := Fn((x^2 - 1)/(x - 1))
```

$$x + 1$$

The field elements are explicit elements of the domain:

```
>> domtype(a)
```

```
Dom::ExpressionField(normal, iszero@normal)
```

Example 2. In the domain `Dom::ExpressionField(id, iszero@normal)` the expressions representing the elements are normalized by the kernel simplifier only:

```
>> Fi := Dom::ExpressionField(id, iszero@normal):
   a := Fi((x^2 - 1)/(x - 1))
```

$$\frac{x^2 - 1}{x - 1}$$

The elements of this domain are not normalized (when viewed as rational expressions over the rationals), thus the domain does not have the axiom `Ax::normalRep`:

```
>> b := a/Fi(x + 1) - Fi(1)
```

$$\frac{x^2 - 1}{(x - 1)\,(x + 1)} - 1$$

But nevertheless this domain also represents the field of rational expressions over the rationals, because zero is detected correctly by the function `iszero @ normal`:

```
>> iszero(b)
```

```
TRUE
```

Changes:

- New methods `"sign"`, `"convert_to"`, `"_less"`, `"_leequal"`, `"rationalize"`, `"solve"`, `"int"` and `"limit"` were implemented.
- The method `"Factor"` disappeared. The methods `"factor"` and `"sqrfree"` now return objects of the domain type `Factored`.
- The method `"equal"` can return `UNKNOWN`.
- If `Normal = id`, then `Dom::ExpressionField` has the axiom `Ax::efficientOperation` for division, multiplication and inversion of elements.
- The domain `Dom::ExpressionField(id,iszero)` is printed as `Dom::ExpressionField(`

`Dom::Float` – the real floating point numbers

`Dom::Float` is the set of real floating point numbers represented by elements of the domain `DOM_FLOAT`.

Creating Elements:

- `Dom::Float(x)`

Parameters:

`x` — an expression which can be converted to a `DOM_FLOAT` by the function `float`.

Categories:

`Cat::DifferentialRing`, `Cat::Field`, `Cat::OrderedSet`

Related Domains: `Dom::Complex`, `Dom::Integer`, `Dom::Numerical`, `Dom::Rational`, `Dom::Real`

Details:

- `Dom::Float` is the domain of real floating point numbers represented by expressions of type `DOM_FLOAT`.
- `Dom::Float` has category `Cat::Field` out of pragmatism. This domain actually is not a field because `bool(1.0 = float(3) / float(3))` returns `FALSE` for example.

- Elements of `Dom::Float` are usually not created explicitly. However, if one creates elements using the usual syntax, it is checked whether the input expression may be converted to a floating point number. This means `Dom::Float` is a facade domain which creates elements of domain type `DOM_FLOAT`.
- Viewed as a differential ring `Dom::Float` is trivial, it contains constants only.
- `Dom::Float` has the domain `Dom::Numerical` as its super domain, i.e., it inherits each method which is defined by `Dom::Numerical` and not implemented by `Dom::Float`. Methods described below are reimplemented by `Dom::Float`.

Entries:

`one`	the unit element; it equals `1.0`.
`zero`	The zero element; it equals `0.0`.

Mathematical Methods

Method `random`: random number generation

```
random()
```

- This methods returns a randomly generated number.

Conversion Methods

Method `convert`: conversion of objects

```
convert(any x)
```

- This method tries to convert `x` to a number of type `Dom::Float`. If the conversion fails, `FAIL` is returned.
- In general, if `float(x)` evaluates to a real floating point number of type `DOM_FLOAT`, this number is the result of the conversion.

Method `convert_to`: conversion to other domains

```
convert_to(dom x, any T)
```

- This method tries to convert the number `x` to an element of type `T`, or, if `T` is not a domain, to the domain type of `T`. If the conversion fails, then `FAIL` is returned.
- The following domains are allowed for `T`: `DOM_FLOAT`, `Dom::Float` and `Dom::Numerical`.

Method `testtype`: type checking

```
testtype(any x, dom T)
```

- This method checks whether it can convert `x` to the domain `Dom::Float`. It returns `TRUE` if it can perform the conversion. Otherwise `FAIL` is returned.
- In general this method is called from the function `testtype` and not directly by the user. Example 2 demonstrates this behavior.

Example 1. Creating some floating point numbers using `Dom::Float`. This example also shows that `Dom::Float` is a facade domain.

```
>> Dom::Float(2.3); domtype(%)

                                2.3

                             DOM_FLOAT

>> Dom::Float(sin(2/3*PI) + 3)

                           3.866025404

>> Dom::Float(sin(x))

 Error: illegal arguments [Dom::Float::new]
```

Example 2. By tracing the method `Dom::Float::testtype` we can see the interaction between `testtype` and `Dom::Float::testtype`.

```
>> prog::trace(Dom::Float::testtype):
   delete x:
   testtype(x, Dom::Float);
   testtype(3.4, Dom::Float);
   prog::untrace(Dom::Float::testtype):

 enter 'Dom::Float::testtype'          with args   : x, Dom::Float
 leave 'Dom::Float::testtype'          with result : FAIL

                             FALSE
 enter 'Dom::Float::testtype'          with args   : 3.4, Dom::Float
 leave 'Dom::Float::testtype'          with result : TRUE

                             TRUE
```

Super-Domain: `Dom::Numerical`

Axioms

`Ax::canonicalRep`, `Ax::systemRep`, `Ax::canonicalOrder`,
`Ax::efficientOperation("_divide")`,
`Ax::efficientOperation("_mult")`,
`Ax::efficientOperation("_invert")`

Changes:

- No changes.

`Dom::Fraction` – the field of fractions of an integral domain

`Dom::Fraction(R)` creates a domain which represents the field of fractions of an integral domain R.

Domain:

- `Dom::Fraction(R)`

Parameters:

`R` — an integral domain, i.e., a domain of category `Cat::IntegralDomain`

Details:

- `Dom::Fraction(R)` creates a domain which represents the field of fractions $F = \left\{ \frac{x}{y} \mid x, y \in R, y \neq 0 \right\}$ of the integral domain R.
- An element of the domain `Dom::Fraction(R)` has two operands, the numerator and denominator.
- If `Dom::Fraction(R)` has the axiom `Ax::canonicalRep` (see below), the denominators have unit normal form and the gcds of numerators and denominators cancel.
- The domain `Dom::Fraction(Dom::Integer)` represents the field of rational numbers. But the created domain is not the domain `Dom::Rational`, because it uses a different representation of its elements. Arithmetic in `Dom::Rational` is much more efficient than it is in `Dom::Fraction(Dom::Integer)`.

Creating Elements:

- `Dom::Fraction(R)(r)`

Parameters:

`r` — a rational expression, or an element of `R`

Categories:

`Cat::QuotientField(R)`

if `R` is a `Cat::DifferentialRing`, then
`Cat::DifferentialRing`

if `R` is a `Cat::PartialDifferentialRing`, then
`Cat::PartialDifferentialRing`

Related Domains: `Dom::Rational`

Details:

- If `r` is a rational expression, then an element of the field of fractions `Dom::Fraction(R)` is created by going through the operands of `r` and converting each operand into an element of `R`. The result of this process is `r` in the form $\frac{x}{y}$, where x and y are elements of `R`. If `R` has `Cat::GcdDomain`, then x and y are coprime.

  If one of the operands can not be converted into the domain `R`, an error message is issued.

Entries:

`characteristic` is the characteristic of `R`.

`coeffRing` is the integral domain `R`.

`one` is the one of the field of fractions of `R`, i.e., the fraction $\frac{1}{1}$.

`zero` is the zero of the field of fractions of `R`, i.e., the fraction $\frac{0}{1}$.

Mathematical Methods

Method `_divide`: divides two fractions

`_divide(`*`dom`* `x, `*`dom`* `y)`

- This method divides the fraction `x` by `y`, i.e., it computes xy^{-1}.
- This method overloads the function `_divide` for fractions, i.e., one may use it in the form `x / y` or in functional notation: `_divide(x, y)`.

Method `_invert`: inverts a fraction

```
_invert(dom r)
```

- This method computes the inverse of $r = \frac{x}{y}$, i.e., it returns $r^{-1} = \frac{y}{x}$ for $x \neq 0$ (if $x = 0$, then an error message is issued).
- This method overloads the function `_invert` for fractions, i.e., one may use it in the form `1/r` or `r^(-1)`, or in functional notation: `_invert(r)`.

Method `_less`: less-than relation

```
_less(dom q, dom r)
```

- This method returns `TRUE` if $q = \frac{x_q}{y_q}$ is smaller than $r = \frac{x_r}{y_r}$, i.e., if $x_q \cdot y_r < x_r \cdot y_q$.
- An implementation is provided only if `R` is an ordered set, i.e., a domain of category `Cat::OrderedSet`.
- This method overloads the function `_less` for fractions, i.e., one may use it in the form `q < r`, or in functional notation: `_less(q, r)`.

Method `_mult`: multiplies fractions by fractions or rational expressions

```
_mult(any q, any r)
```

- If `q` and `r` are both fractions of the same type, the product $q \cdot r$ is computed directly. The resulting fraction is normalized (see the methods `"normalize"` and `"normalizePrime"`).
- If `q` is not of the domain type `Dom::Fraction(R)`, it is considered as a rational expression which is converted into a fraction over `R` and multiplied with `q`. If the conversion fails, `FAIL` is returned. The same applies to `r`.
- This method also handles more than two arguments. In this case, the argument list is splitted into two parts of the same length which both are multiplied with the function `_mult`. The two results are multiplied again with `_mult` whose result then is returned.
- This method overloads the function `_mult` for fractions, i.e., one may use it in the form `q * r` or in functional notation: `_mult(q, r)`.

Method `_negate`: negates a fraction

```
_negate(dom r)
```

- This method computes $-r$ by negating the numerator of r.
- This method overloads the function `_negate` for fractions, i.e., one may use it in the form `-r` or in functional notation: `_negate(r)`.

Method `_power`: the integer power of a fraction

```
_power(dom r, integer n)
```

- This method computes r^n for integers `n`.
- This method overloads the function `_power` for fractions, i.e., one may use it in the form `r^n` or in functional notation: `_power(r, n)`.

Method `_plus`: adds fractions

```
_plus(dom q, dom r, ...)
```

- Returns the sum $q + r + \cdots$ of fractions. The returned fraction is normalized (see the methods `"normalize"` and `"normalizePrime"`).
- If one of the arguments is not of the domain type `Dom::Fraction(R)`, then `FAIL` is returned.
- This method overloads the function `_plus` for fractions, i.e., one may use it in the form `q + r` or in functional notation: `_plus(q, r)`.

Method `D`: the differential operator

```
D(dom r)
```

- This method takes the derivative of the fraction $r = \frac{x}{y}$, i.e., it returns the fraction $\frac{D(x)y - xD(y)}{y^2}$.
  The resulting fraction is normalized (see the methods `"normalize"` and `"normalizePrime"`).
- An implementation is provided only if `R` is a partial differential ring, i.e., a domain of category `Cat::PartialDifferentialRing`.
- This method overloads the operator `D` for fractions, i.e., one may use it in the form `D(r)`.

Method `denom`: the denominator of a fraction

```
denom(dom r)
```

- This method returns the denominator of `r`, an element of the integral domain `R`.
- This method overloads the function `denom` for fractions, i.e., one may use it in the form `denom(r)`.

Method `diff`: differentiation of fractions

```
diff(dom r, variable u)
```

- This method returns the fraction which results when differentiating the fraction $r = \frac{x}{y}$, i.e., it returns the fraction $\left(\frac{\partial x}{\partial u}y - x\frac{\partial y}{\partial u}\right)/y^2$.
  The resulting fraction is normalized (see the methods `"normalize"` and `"normalizePrime"`).
- This method overloads the function `diff` for fractions, i.e., one may use it in the form `diff(r, u)`.
- An implementation is provided only if `R` is a partial differential ring, i.e., a domain of category `Cat::PartialDifferentialRing`.

Method `equal`: test on equality of fractions

```
equal(dom q, dom r)
```

- This method tests if the fraction `q` is equal to `r`, and returns `TRUE`, `FALSE` or `UNKNOWN`, respectively.

Method `factor`: factorizes the numerator and denominator of a fraction

```
factor(dom r)
```

- This method factorizes the numerator and denominator of `r` into irreducible factors and returns `r` in the form $r = u \cdot r_1^{e_1} \cdot \ldots \cdot r_n^{e_n}$.
  The result is a factored object, i.e., an element of the domain type `Factored`. Its factorization type is `"irreducible"` and the factorization ring is `R`.
- The factors $u, r_1, \ldots, r_n$ are fractions of type `Dom::Fraction(R)`, the exponents $e_1, \ldots, e_n$ are integers.
- The system function `factor` is used to perform the factorization of the numerator and denominator of `r`.
- This method overloads the function `factor` for fractions, i.e., one may use it in the form `factor(r)`.

Method `intmult`: integer multiple of a fraction

```
intmult(dom r, integer n)
```

- This method computes nr.

Method `iszero`: test for zero

```
iszero(dom r)
```

- This method returns `TRUE` if `r` is zero, and `FALSE` otherwise.
- An element of the field `Dom::Fraction(R)` is zero if its numerator is the zero element of `R`. Note that there may be more than one representation of the zero element if `R` does not have `Ax::canonicalRep`.
- This method overloads the function `iszero` for fractions, i.e., one may use it in the form `iszero(r)`.

Method `numer`: the numerator of a fraction

```
numer(dom r)
```

- This method returns the numerator of `r`, an element of the integral domain `R`.
- This method overloads the function `numer` for fractions, i.e., one may use it in the form `numer(r)`.

Method `random`: random fraction generation

```
random()
```

- This method returns a randomly generated fraction. It uses the method `"random"` of the domain `R` to randomly generate its numerator and denominator.
- The returning fraction is normalized (see the methods `"normalize"` and `"normalizePrime"`.

Conversion Methods

Method `convert_to`: fraction conversion

```
convert_to(dom r, any T)
```

- This method tries to convert `r` into an element of the domain `T`, or, if `T` is not a domain, to the domain type of `T`.
- If the conversion fails, `FAIL` is returned.
- The conversion succeeds if `T` is one of the following domains: `Dom::Expression` or `Dom::ArithmeticalExpression`.
- Use the function `expr` to convert `r` into an object of a kernel domain (see below).

Method `expr`: converts a fraction into an object of a kernel domain

```
expr(dom r)
```

- This method converts `r` into an expression by converting numerator and denominator into expressions (using the method `"expr"` of `R`).
- The result is an object of a kernel domain (e.g., `DOM_RAT` or `DOM_EXPR`).
- This method overloads the function `expr` for fractions, i.e., one may use it in the form `expr(r)`.

Method `TeX`: TeX formatting of a fraction

```
TeX(dom r)
```

- This method returns a TEX-formatted string for the fraction `r` in form of a TEX `\frac` construct.
- The method `TeX` of the component ring `R` is used to get the TEX-representations of the numerator and denominator of `r`, respectively.

Method `retract`: retraction to base domain

```
retract(dom r)
```

- This method divides the numerator of `r` by the denominator of `r` and returns the result, if it is an element of `R`. Otherwise, `FAIL` is returned.

Technical Methods

Method `normalize`: normalizing fractions

```
normalize(R x, R y)
```

- This method normalizes the fraction $\frac{x}{y}$ which then is returned.
- Normalization means to remove the gcd of `x` and `y`. Hence, `R` needs to be of category `Cat::GcdDomain`. Otherwise, normalization cannot be performed and the result of this method is the fraction $\frac{x}{y}$.

Method `normalizePrime`: normalizing fractions over integral domains with a gcd

```
normalizePrime(R x, R y)
```

- This method returns the fraction $\frac{x}{y}$. If `x` is zero, the fraction is normalized to zero.

- In rings of category `Cat::GcdDomain`, elements are assumed to be relatively prime. Hence, there is no need to normalize the fraction $\frac{x}{y}$.
- In rings not of category `Cat::GcdDomain`, normalization of elements can not be performed and the result of this method is the fraction $\frac{x}{y}$.

Example 1. We define the field of rational functions over the rationals:

```
>> F := Dom::Fraction(Dom::Polynomial(Dom::Rational))
```

```
     Dom::Fraction(Dom::Polynomial(Dom::Rational, LexOrder))
```

and create an element of `F`:

```
>> a := F(y/(x - 1) + 1/(x + 1))
```

```
                          x + y + x y - 1
                          ---------------
                                2
                               x  - 1
```

To calculate with such elements use the standard arithmetical operators:

```
>> 2*a, 1/a, a*a
```

```
                                  2
 2 x + 2 y + 2 x y - 2          x  - 1
 ---------------------, ---------------,
         2                x + y + x y - 1
        x  - 1

                    2    2        2      2      2  2
    - 2 x - 2 y + x  + y  + 2 x y  + 2 x  y + x  y  + 1
    ------------------------------------------------
                          2    4
                      - 2 x  + x  + 1
```

Some system functions are overloaded for elements of domains generated by `Dom::Fraction`, such as `diff`, `numer` or `denom` (see the description of the corresponding methods `"diff"`, `"numer"` and `"denom"` above).

For example, to differentiate the fraction `a` with respect to `x` enter:

```
>> diff(a, x)
```

```
                                 2    2
                2 x - y - 2 x y - x  - x  y - 1
                -------------------------------
                             2    4
                         - 2 x  + x  + 1
```

If one knows the variables in advance, then using the domain `Dom::DistributedPolynomial` yields a more efficient arithmetic of rational functions:

```
>> Fxy := Dom::Fraction(
     Dom::DistributedPolynomial([x, y], Dom::Rational)
   )

 Dom::Fraction(Dom::DistributedPolynomial([x, y],

    Dom::Rational, LexOrder))

>> b := Fxy(y/(x - 1) + 1/(x + 1)):
   b^3

                          2    3      2    3        2      2
 (3 x + 3 y - 3 x y - 3 x  + x  - 3 y  + y  - 3 x y  - 3 x  y +

         3      3        2  2      2  3      3  2    3  3
    3 x y  + 3 x  y + 3 x  y  + 3 x  y  + 3 x  y  + x  y  -
1) /

        2      4    6
    (3 x  - 3 x  + x  - 1)
```

Example 2. We create the field of rational numbers as the field of fractions of the integers, i.e., $\mathbb{Q} = \left\{ \frac{x}{y} \mid x, y \in \mathbb{Z} \right\}$:

```
>> Q := Dom::Fraction(Dom::Integer):
   Q(1/3)

                                 1/3

>> domtype(%)

                  Dom::Fraction(Dom::Integer)
```

Another representation of $\mathbb{Q}$ in MuPAD is the domain `Dom::Rational` where the rationals are of the kernel domains `DOM_INT` and `DOM_RAT`. Therefore it is much more efficient to work with `Dom::Rational` than with `Dom::Fraction(Dom::Integer)`.

Super-Domain: `Dom::BaseDomain`

Axioms

`Ax::normalRep`

if R has `Ax::canonicalRep`
if R is a `Cat::GcdDomain`
if R has `Ax::canonicalUnitNormal`
`Ax::canonicalRep`

Changes:

- New method `"factor"` for factoring fractions.

`Dom::GaloisField` – finite fields

`Dom::GaloisField(p, n, f)` creates the residue class field $Z_p[X]/\langle f\rangle$, a finite field with p^n elements. If `f` is not given, it is chosen at random among all irreducible polynomials of degree n.

`Dom::GaloisField(q)` (where $q = p^n$) is equivalent to `Dom::GaloisField(p,n)`.

`Dom::GaloisField(F, n, f)` creates the residue class field $F[X]/\langle f\rangle$, a finite field with $|F|^n$ elements. If `f` is not given, it is chosen at random among all irreducible polynomials of degree n.

Domain:

- `Dom::GaloisField(q)`
- `Dom::GaloisField(p, n)`
- `Dom::GaloisField(p, n, f)`
- `Dom::GaloisField(F, n)`
- `Dom::GaloisField(F, n, f)`

Parameters:

`q` — prime power
`p` — prime
`n` — positive integer
`f` — univariate irreducible polynomial over `Dom::IntegerMod(p)` or `F`, or polynomial expression convertible to such
`F` — finite field of type `Dom::IntegerMod` or `Dom::GaloisField`.

Details:

- If `f` is not given, a random irreducible polynomial of appropriate degree is used; some free identifier is chosen as its variable, and this one must also be used when creating domain elements.
- Although $n = 1$ is allowed, `Dom::IntegerMod` should be used for representing prime fields.
- If `F` is of type `Dom::GaloisField`, consisting of residue classes of polynomials, the variable of these polynomials must be distinct from the variable of `f`. If a tower several of Galois fields is constructed, the variable used in the uppermost Galois field must not equal any of those used in

the tower. A special entry `"VariablesInUse"` serves to keep track of all variables appearing somewhere in the tower.

`Dom::GaloisField(p,n,f)(g)` (or, respectively, `Dom::GaloisField(F,n,f)(g)`) creates the residue class of `g` modulo `f`. It is represented by the unique polynomial in that class that has smaller degree than `f`.

Creating Elements:

- `Dom::GaloisField(p, n, f)(g)`

Parameters:

`g` — univariate polynomial over the ground field in the same variable as `f`, or polynomial expression convertible to such

Categories:

`Cat::Field`, `Cat::Algebra(F)`, `Cat::VectorSpace(F)`

Related Domains: `Dom::AlgebraicExtension`, `Dom::IntegerMod`

Entries:

`zero` the zero element of the field

`one` the unit element of the field

`characteristic` the characteristic of the field

`size` the number of elements of the field

`PrimeField` the prime field, which equals `Dom::IntegerMod(p)`.

`Variable` the variable of the polynomial `f`.

`VariablesInUse` a list consisting of `"Variable"` and the variables used by the ground field.

`companionMatrix` an n times n–matrix over the ground field, where n is the degree of the field over its ground field. It can be used for representing field elements as matrices since its minimal polynomial is `f`.

`companionPowers` a list of the first $n-1$ powers of the companion matrix.

Mathematical Methods

Method `iszero`: test for zero

```
iszero(dom a)
```

- This method returns TRUE if a equals the zero element of the field, and FALSE if not.
- It overloads the function `iszero`.

Method `_power`: integer power of an element

```
_power(dom a, integer n)
```

- This method computes a^n.
- It overloads `_power`.

Method `frobenius`: Frobenius map

```
frobenius(dom a)
```

- This method computes a^p, where p is the size of the ground field.

Method `conjugates`: conjugates of an element

```
conjugates(dom a)
```

- This method computes the list of all distinct conjugates of a.

Method `order`: order of an element

```
order(dom a)
```

- This method computes the smallest postive integer n such that $a^n = 1$.

Method `isSquare`: test whether an element is a square

```
isSquare(dom a)
```

- This method returns TRUE if there exists an x in the field such that $x^2 = a$, and FALSE otherwise.

Method `ln`: discrete logarithm

```
ln(dom a, dom b)
```

- This method returns the smallest positive integer n such that $b^n = a$, or `infinity` if such an integer does not exist. `b` must be nonzero.

Method `elementNumber`: enumerate field elements

```
elementNumber(dom a)
```

- This method assigns to every element $\sum_{i=0}^{n-1} a_i x^i$ the number $\sum_{i=0}^{n-1} \phi(a_i) q^i$, where q is the number of elements of the ground field, and ϕ assigns to each element of the ground field a number between 0 and $q-1$: if the ground field is a `Dom::GaloisField` itself, its method `elementNumber` is used; if the ground field is a prime field, ϕ is taken to be the mapping that assigns, to each residue class modulo q, its smallest nonnegative member.
- The inverse of this mapping has not been implemented.

Method `matrixRepresentation`: isomorphism to the algebra generated by the companion matrix

```
matrixRepresentation(dom a)
```

- This method implements the unique isomorphism between the field $F[X]/\langle f \rangle$ and the algebra of all linear combinations of powers of the companion matrix that maps X to the companion matrix.
- If A is the companion matrix, the image of $\sum_i a_i X^i$ is $\sum_i a_i A^i$.

Method `randomPrimitive`: choose a primitive element at random

```
randomPrimitive()
```

- This method returns a randomly chosen primitive element of the field. The result is uniformly distributed.

Method `isBasis`: tests elements for being a basis over the ground field

```
isBasis(list of dom l)
```

- This method tests whether the field elements in the list `l` constitute a basis of the field, viewed as vector space over the ground field.

Method `isNormal`: tests whether a given field element is normal

```
isNormal(dom a)
```

- Let n be the degree of the field over its ground field. This method tests whether the powers $a^0 \ldots a^{n-1}$ form a base of the field over its ground field. If this is the case, `TRUE` is returned; otherwise the result is `FALSE`.

Method `randomNormal`: choose normal element at random

```
randomNormal()
```

- This method chooses a random normal element simply by choosing random elements until a normal one is found.

Method `isPrimitivePolynomial`: tests whether a polynomial over the field is primitive

```
isPrimitivePolynomial(univariate polynomial over dom h)
```

- Let G be the field represented by *dom*. If `h` is a polynomial in z, this method returns `TRUE` if `h` is irreducible and all nonzero elements of the extension field $G[z]/\langle h \rangle$ are powers of z. Otherwise, `FALSE` is returned.

Conversion Methods

Method `convert`: conversion from other types

```
convert(any a)
```

- The object `a` can be converted to an element of the Galois field exactly if it can be converted to an element of the super-domain, `Dom::AlgebraicExtension`.

Method `convert_to`: conversion to other types

```
convert_to(dom a, domain T)
```

- returns an object of type `T` if `a` can converted to that type, or `FAIL` otherwise. In the current version, only conversions to the type `DOM_POLY` are possible.

Example 1. We define `L` to be the field with 4 elements. Then $a^4 = a$ for every $a \in L$, by a well-known theorem.

```
>> L:=Dom::GaloisField(2, 2, u^2+u+1): L(u+1)^4
                                  u + 1
```

Super-Domain: `Dom::AlgebraicExtension`

Axioms

`Ax::canonicalRep`

Changes:

- No changes.

`Dom::Ideal` – the domains of sets of ideals

`Dom::Ideal(R)` creates the domain of finitely generated ideals of the ring `R`.

Domain:

- `Dom::Ideal(R)`

Parameters:

`R` — domain of category `Cat::Ring`

`Dom::Ideal(R)([a1, ...,an])` or `Dom::Ideal(R)({a1, ...,an})` creates the ideal generated by the elements `a1` through `an`.

Creating Elements:

- `Dom::Ideal(R)([a1, ...])`
- `Dom::Ideal(R)({a1, ...})`

Parameters:

`a1, ...` — elements of `R`

Categories:

`Cat::Monoid`

Entries:

`coeffRing`	the ring `R`
`zero`	the ideal consisting only of the zero element of `R`.
`one`	the ideal generated by `R::one`, i.e., `R` itself.

Mathematical Methods

Method `iszero`: tests whether an ideal is zero

```
iszero(dom J)
```

- This method returns `TRUE` if `J` is the zero ideal, and `FALSE` otherwise.
- It overloads the function `iszero`.

Method `_mult`: product of ideals

```
_mult(dom J1, ...)
```

- The product of ideals $J_1 \cdot \cdots \cdot J_k$ is defined to be the ideal generated by all products $a_1 \cdot \cdots \cdot a_k$, where $a_i \in J_i$.
- This method overloads the function `_mult`.

Method `_plus`: sum of ideals

```
_plus(dom J1, ...)
```

- The sum of ideals $J_1 + \ldots + J_k$ is defined to be the ideal of all sums $a_1 + \ldots + a_k$, where $a_i \in J_i$.
- This method overloads the function `_plus`.

Method `_negate`: negate an ideal

```
_negate(dom J)
```

- Since -1 is a unit in every ring, negating an ideal just gives the ideal itself.
- This method overloads the function `_negate`.

Method `_subtract`: difference of ideals

```
_subtract(dom J1, dom J2)
```

- This is the same as $J_1 + J_2$ since $J_2 = -J_2$.
- This method overloads the function `_subtract`.

Method `normal`: normal form of an ideal

```
normal(dom J)
```

- This method only exists for ideals over polynomial rings over fields; in this case, it returns a Gröbner base for the ideal.
- It overloads the function `normal`.

Conversion Methods

Method `convert`: convert list or set to ideal

```
convert(list or set of ring elements l)
```

- returns the ideal generated by the elements of `l`.

Method `expr`: list of generators of an ideal

```
expr(dom J)
```

- This method returns the list of generators defining `J`.
- It overloads the function `expr`.

Example 1. We define `R` to be the polynomial ring $Q[x, y, z]$.

```
>> R:=Dom::DistributedPolynomial([x,y,z], Dom::Rational)

 Dom::DistributedPolynomial([x, y, z], Dom::Rational, LexOrder)
```

Next, we define an ideal `J` over `R` by giving a list of generators.

```
>> J:=Dom::Ideal(R)([x*y+y^2*x+x*y+z+1, z^2-x*z-y*x-7])

        <[x*y^2 + 2*x*y + z + 1, - x*y - x*z + z^2 - 7]>
```

Since `R` is a polynomial ring over a field, a Gröbner base of `J` can be obtained as follows:

```
>> normal(J)

 <[z - 13*y + y*z - 7*y^2 + z^2 + 2*y*z^2 + y^2*z^2, 8*z -
7*y \
 - 2*x*z + 2*z^2 - z^3 + x*z^2 + y*z^2 - 13, x*y + x*z - z^2 + \
 7]>
```

Super-Domain: `Dom::BaseDomain`

Changes:

- `Dom::Ideal` is a new domain.

`Dom::ImageSet` – the domain of images of sets under mappings

`Dom::ImageSet` is the domain of all sets of complex numbers that can be written as the set of all values taken on by some mapping, i.e., sets of the form $\{f(x1, \ldots, x_n); x_i \in S_i\}$ for some function f and some sets $S_1, \ldots, S_n$.

Domain:

- `Dom::ImageSet`

Details:

- Image sets are mainly used by `solve` to express sets like $\{k * \pi; k \in Z\}$.
- `Dom::ImageSet` belongs to the category `Cat::Set`—arithmetical and set-theoretic operations are inherited from there.

`Dom::ImageSet(f, x, S)` represents the set of all values that can be obtained by substituting some element of `S` for `x` in the expression `f`.

`Dom::ImageSet(f, [x1, ...], [S1, ...])` represents the set of all values that can be obtained by substituting, for each i, the identifier x_i by some element of S_i in the expression `f`.

Creating Elements:

- `Dom::ImageSet(f, x, S)`
- `Dom::ImageSet(f, [x1, ...], [S1, ...])`

Parameters:

`f` — arithmetical expression
`x` — identifier or indexed identifier
`S` — set of any type

Categories:

`Cat::Set`

Details:

- See `solve` for an overview of the different kinds of sets in MuPAD.
- If a list of several identifiers is given, the identifiers must be distinct.

Mathematical Methods

Method `changevar`: change the name of a variable

`changevar(`*`dom`* `A, `*`identifier`* `oldvar, `*`identifier`* `newvar)`

- replaces `oldvar` by `newvar` both in the expression and in the list of variables. This gives (mathematically) the same set, since $\{f(x); x \in S\} = \{f(y); y \in S\}$.
- The new variable `newvar` must not equal any element of the list of variables; this is not checked!

Method `setvar`: set the name of the variable

`setvar(`*`dom`* `A, `*`identifier`* `newvar)`

- The only variable of `A` is replaced by `newvar` both in the expression and in the (one-element) list of variables. This method may only be applied for image sets in one variable.

`setvar(`*`any`* `A, `*`identifier`* `newvar)`

- For an argument `A` that is not an image set, the method `"setvar"` is applied to all image sets contained in the expression `A`. `A` might be, for example, a union, intersection, etc. of image sets and other sets.

Method `homogpointwise`: define an n-ary pointwise operator for image sets

`homogpointwise(`*`any`* `Op)`

- This method returns a procedure which implements a continuation of the function `Op`. `Op` must be a function that maps each finite sequence of (arbitrarily many) complex numbers to a single complex number (e.g. their sum or product). `Op` is set forth to the class of image sets by defining `Op(A1,...,An)` to be the set of all `Op(a1,..,an)`, where $a_i \in A_i$ for each i.
- `Op` must accept arithmetical expressions as arguments.

Method `isEmpty`: tests whether a set is empty

`isEmpty(`*`dom`* `A)`

- A is empty if and only if one of its parameters ranges over the empty set. This method tries to decide whether this is the case and returns `TRUE`, `FALSE`, or `UNKNOWN`.

Method `substituteBySet`: substitute an ImageSet for a variable

`substituteBySet(`*`arithmetical expression`* `a,` *`identifier`* `x,` *`dom`* `A)`

- This method returns the set of all numbers that can be obtained by substituting some element of `A` for `x` in the expression `a`. That is, viewing $a = a(x)$ as a function $C \mapsto C$, this method returns $\{a(x); x \in A\}$, the image of the restriction of that function to `A`.

Method `indets`: free parameters of a set

`indets(`*`dom`* `A)`

- This method returns the set of free parameters the set `A` depends on.
- If $A = \{f(x_1, \ldots, x_n, y_1, \ldots y_k); x_i \in S_i\}$, the x_i are called bound and the y_i are called free parameters.
- Use the slot `"variables"` to obtain the bound parameters.
- This method overloads the function `indets`.

Access Methods

Method `expr`: the defining mapping as an expression

`expr(`*`dom`* `A)`

- If $A = \{f(x); x \in S\}$, this method returns the expression `f`; similarly if several variables range over several sets.
- This method overloads the function `expr`.

Method `variables`: list of variables

`variables(`*`dom`* `A)`

- If $A = \{f(x_1, .., x_n); x_i \in S_i\}$, this method returns the list of variables `[x1,...,xn]`.
- The free parameters (identifiers appearing in f other than the x_i) can be obtained using `indets`.

Method `nvars`: number of variables

`nvars(`*dom* `A)`

- If $A = \{f(x_1, .., x_n); x_i \in S_i\}$, this method returns the number n of variables.

Method `sets`: list of sets

`sets(`*dom* `A)`

- If $A = \{f(x_1, .., x_n); x_i \in S_i\}$, this method returns the list of sets `[S1,...,Sn]`.

Technical Methods

Method `print`: print image set

`print(`*dom* `A)`

- This method returns a string used for displaying `A` on the screen.

Example 1. We define S to be the set of all integer multiples of π.

```
>> S:=Dom::ImageSet(k*PI, k, Z_)

                      { X1*PI |  X1 in Z_ }
```

We may now apply the usual set–theoretic operations.

```
>> S intersect Dom::Interval(3..7)

                             {PI, 2 PI}
```

Super-Domain: `Dom::BaseDomain`

Changes:

- `Dom::ImageSet` is a new domain.

`Dom::Integer` – the ring of integer numbers

`Dom::Integer` is the ring of integer numbers represented by elements of the domain `DOM_INT`.

Creating Elements:

- `Dom::Integer(x)`

Parameters:

`x` — an integer

Categories:

`Cat::EuclideanDomain`, `Cat::FactorialDomain`, `Cat::DifferentialRing`, `Cat::OrderedSet`

Related Domains: `Dom::Complex`, `Dom::Float`, `Dom::Numerical`, `Dom::Rational`, `Dom::Real`

Details:

- `Dom::Integer` is the domain of integer numbers represented by expressions of type `DOM_INT`.
- Elements of `Dom::Integer` are usually not created explicitly. However, if one creates elements using the usual syntax, it is checked whether the input is an integer number. This means that `Dom::Integer` is a façade domain which creates elements of domain type `DOM_INT`.
- Viewed as a differential ring `Dom::Integer` is trivial, it contains constants only.
- `Dom::Integer` has the domain `Dom::Numerical` as its super domain, i.e., it inherits each method which is defined by `Dom::Numerical` and not re-implemented by `Dom::Integer`. Methods described below are those implemented by `Dom::Integer`.

Mathematical Methods

Method `associates`: associate elements

`associates(`*`dom`* `x, `*`dom`* `y)`

- This method returns `TRUE` if `x` and `y` are associates, i.e., if `abs(x) = abs(y)`, otherwise it returns `FALSE`.

Method `_divide`: division of two objects

`_divide(`*`dom`* `x, `*`dom`* `y)`

- This method returns `x/y` if `y` divides `x` and `FAIL` otherwise.

Method `_divides`: decide if a number divides another one

```
_divides(dom x, dom y)
```

- This method returns `TRUE` if `x` divides `y`

Method `euclideanDegree`: Euclidean degree

```
euclideanDegree(dom x)
```

- This method returns the Euclidean degree of `x`, i.e., `abs(x)`.

Method `factor`: factorization

```
factor(dom x)
```

- This method returns the factorization of `x`. The result of this method is an object of the domain `Factored`.

Method `gcd`: gcd computation

```
gcd(dom x1, dom x2, ...)
```

- This method returns the gcd of the given arguments `x1, x2, ...`

Method `gcdex`: applies the extended Euclidean algorithm

```
gcdex(dom x, dom y)
```

- This method returns g, v, w such that $g = gcd(x, y) = x * v + y * w$.

Method `_invert`: inverse of an element

```
_invert(dom x)
```

- This method returns the multiplicative inverse of `x`. This is `x` if `x` is `1` or `-1`, and `FAIL` otherwise.

Method `irreducible`: prime number test

```
irreducible(dom x)
```

- This method returns `TRUE` if and only if `x` is prime.

Method `isUnit`: tests if an element is a unit

```
isUnit(dom x)
```

- This method returns `TRUE` if `x` is a unit, i.e., if `x` is `1` or `-1`, otherwise it returns `FALSE`.

Method `lcm`: computes the lcm

```
lcm(dom x1, dom x2, ...)
```

- This method returns the lcm of the given arguments `x1, x2, ...`

Method `quo`: computes the euclidean quotient

```
quo(dom x, dom y)
```

- This method returns the euclidean quotient of `x` and `y`, i.e., `x div y`.

Method `random`: random number generation

```
random()
```

- This methods returns a random integer number between -999 and 999.

```
random(integer n)
```

- This methods returns a random number between 0 and $\texttt{n} - 1$.

```
random(integer m..integer n)
```

- This methods returns a random number between `m` and `n`.

Method `rem`: computes the Euclidean reminder

```
rem(dom x, dom y)
```

- This method returns the Euclidean reminder of `x` and `y`, i.e., `modp(x,y)`.

Method `unitNormal`: unit normal part

```
unitNormal(dom x)
```

- This method returns the unit normal part of `x`, i.e., `abs(x)`.

Method `unitNormalRep`: unit normal representation

```
unitNormalRep(dom x)
```

- This method returns the unit normal representation of `x`, i.e., the list `[abs(x), sign(x), -sign(x)]`.

Conversion Methods

Method `convert`: conversion of objects

`convert(`*`any`* `x)`

- This method tries to convert `x` to a integer of type `Dom::Integer`. This is only possible if `x` is of type `DOM_INT`. If the conversion fails, `FAIL` is returned.

Method `convert_to`: conversion to other domains

`convert_to(`*`dom`* `x,` *`any`* `T)`

- This method tries to convert the number `x` to an element of type `T`, or, if `T` is not a domain, to the domain type of `T`. If the conversion fails, then `FAIL` is returned.
- The following domains are allowed for for `T`: `DOM_INT`, `Dom::Integer`, `Dom::Rational`, `DOM_FLOAT`, `Dom::Float` and `Dom::Numerical`.

Method `testtype`: type checking

`testtype(`*`any`* `x,` *`dom`* `T)`

- This method checks whether it can convert `x` to the domain `Dom::Integer`. This is the case if `x` is of type `DOM_INT`. It returns `TRUE` if it can perform the conversion. Otherwise `FAIL` is returned.
- Usually, this method is called from the function `testtype` and not directly by the user. Example 2 demonstrates this behavior.

Example 1. Creating some integer numbers using `Dom::Integer`. This example also shows that `Dom::Integer` is a façade domain.

```
>> Dom::Integer(2); domtype(%)

                                    2

                                DOM_INT

>> Dom::Integer(2/3)

 Error: illegal arguments [Dom::Integer::new]
```

Example 2. By tracing the method `Dom::Integer::testtype` we can see the interaction between `testtype` and `Dom::Integer::testtype`.

```
>> prog::trace(Dom::Integer::testtype):
   delete x:
   testtype(x, Dom::Integer);
   testtype(3, Dom::Integer);
   prog::untrace(Dom::Integer::testtype):

 enter 'Dom::Integer::testtype'         with args   : x, Dom::Integer
 leave 'Dom::Integer::testtype'         with result : FAIL

                                FALSE
 enter 'Dom::Integer::testtype'         with args   : 3, Dom::Integer
 leave 'Dom::Integer::testtype'         with result : TRUE

                                TRUE
```

Super-Domain: `Dom::Numerical`

Axioms

`Ax::canonicalRep`, `Ax::systemRep`, `Ax::canonicalOrder`, `Ax::canonicalUnitNormal`, `Ax::closedUnitNormals`, `Ax::efficientOperation("_divide")`, `Ax::efficientOperation("_mult")`

Changes:

- No changes.

`Dom::IntegerMod` – residue class rings modulo integers

`Dom::IntegerMod(n)` creates the residue class ring of integers modulo `n`.

Domain:

- `Dom::IntegerMod(n)`

Parameters:

`n` — positive integer greater than 1

`Dom::IntegerMod(n)(a)` creates the residue class of `a` modulo `n`.

Creating Elements:

- `Dom::IntegerMod(n)(a)`

Parameters:

`a` — any integer or a rational number whose denominator is coprime to `n`

Categories:

if `n` has `Type::Prime`
`Cat::Field`

else
`Cat::CommutativeRing`

Related Domains: `Dom::Integer`, `Dom::GaloisField`

Entries:

`characteristic`	the characteristic of the residue class ring, `n`
`one`	the unit element, `1 mod n`
`zero`	the zero element, `0 mod n`

Mathematical Methods

Method `_divide`: division of two elements

`_divide(`*`dom`* `element1, `*`dom`* `element2)`

- This method divides two elements. The result is an element of the residue class ring.
- This method overloads `_divide`.

Method `_invert`: invert elements

`_invert(`*`dom`* `element)`

- This method inverts an element. The result is an element of the residue class ring.
- This method overloads `_invert`.

Method `_mult`: multiply elements

`_mult(`*`dom`* `element, ...)`

- This method multiplies elements. The result is an element of the residue class ring.
- This method overloads `_mult`.

Method `_negate`: negate elements

```
_negate(dom element)
```

- This method negates an element. The result is an element of the residue class ring.
- This method overloads `_negate`.

Method `_plus`: add elements

```
_plus(dom element, ...)
```

- This method adds elements. The result is an element of the residue class ring.
- This method overloads `_plus`.

Method `_power`: power of elements

```
_power(dom element, integer power)
```

- This method returns the `power`th power of an element. The result is an element of the residue class ring.
- This method overloads `_power`.

Method `_subtract`: subtraction of two elements

```
_subtract(dom element1, dom element2)
```

- This method subtracts two elements. The result is an element of the residue class ring.
- This method overloads `_subtract`.

Method `isSquare`: test for being a square

```
isSquare(dom element)
```

- This method returns `TRUE` if `element` is the square of another element, and `FALSE` otherwise.

Method `iszero`: zero test

```
iszero(dom element)
```

- This method returns `TRUE`, if `element` is zero, otherwise `FALSE`.
- This method overloads `iszero`.

Method `ln`: discrete logarithm

`ln(`*`dom`* `element, `*`dom`* `base)`

- This method returns the discrete logarithm of `element` with respect to the base `base`.
- The result is `infinity` if `element` is not in the subgroup generated by `base`.
- The result is `FAIL` if `base` is not a unit.
- This method overloads `ln`.

Method `order`: order

`order(`*`dom`* `element)`

- This method returns the order of `element` in the group of multiplicative units.
- The result is `FAIL` if `element` is not a unit.

Conversion Methods

Method `TeX`: TeX output

`TeX(`*`dom`* `element)`

- This method converts `element` to a TeX-formatted string.

Method `convert`: conversion

`convert(`*`Type::Rational`* `number)`

- This method converts an integer or rational number into an element of the domain.
- The conversion fails if the denominator of `number` and the modulus `n` are not relatively prime.

Method `convert_to`: conversion

`convert_to(`*`dom`* `element, `*`DOM_DOMAIN`* `domain)`

- This method converts `element` into an element of the given domain if possible (now `DOM_INT` and `Dom::Integer`).

Method `expr`: convert an element to an expression

`expr(`*`dom`* `element)`

- This method converts `element` into an integer number.
- This method overloads `expr`.

Technical Methods

Method `print`: printing elements

`print(`*`dom`* `element)`

- This method returns an expression used for displaying the element.

Method `random`: random element

`random()`

- This method creates a random element of the domain.
- This method overloads `random`.

Example 1. We define the residue class ring $\mathbb{Z}_7$:

```
>> Z7:= Dom::IntegerMod(7)
                          Dom::IntegerMod(7)
```

Next, we create some elements:

```
>> a:= Z7(1); b:= Z7(2); c:= Z7(3)
                                1 mod 7

                                2 mod 7

                                3 mod 7
```

We may use infix notation for arithmetical operations since the operators have been overloaded:

```
>> a + b, a*b*c, 1/c, b/c/a/c
                 3 mod 7, 6 mod 7, 5 mod 7, 1 mod 7
```

`a` and `b` are squares while `c` is not:

```
>> Z7::isSquare(a), Z7::isSquare(b), Z7::isSquare(c)
                           TRUE, TRUE, FALSE
```

Indeed, `c` is a generator of the group of units:

```
>> Z7::order(a), Z7::order(b), Z7::order(c)
                                1, 3, 6
```

Super-Domain: `Dom::BaseDomain`

Axioms

`Ax::normalRep`, `Ax::canonicalRep`, `Ax::noZeroDivisors`,
`Ax::closedUnitNormals`, `Ax::canonicalUnitNormal`,
`Ax::efficientOperation("_invert")`,
`Ax::efficientOperation("_divide")`,
`Ax::efficientOperation("_mult")`

Changes:

- No changes.

`Dom::Interval` – intervals of real numbers

`Dom::Interval` represents the set of all intervals of real numbers.

`Dom::Interval(l, r)` creates the interval of all real numbers between `l` and `r`. If a border is given as a list with `l` or `r` as the sole element, this border will be regarded as a closed border, otherwise the interval does not contain `l` and `r`.

A border can be any arithmetical expression that could represent a real number, e.g., `sqrt(2*x)` and `a + I`. Properties are ignored.

Creating Elements:

- `Dom::Interval(l, r)`
- `Dom::Interval([l], r)`
- `Dom::Interval(l, [r])`
- `Dom::Interval([l], [r])`
- `Dom::Interval([l, r])`

Parameters:

`l` — The left border. If given as a list of one element (the left border), the interval will be created as left closed.

`r` — The right border. If given as a list of one element (the right border), the interval will be created as right closed.

Categories:

`Cat::Set`, `Cat::AbelianMonoid`

Related Domains: `Type::Interval`

Details:

- `Dom::Interval` creates real intervals. The domain `Dom::Interval` provides fundamental operations to combine intervals with intervals and other mathematical objects.
- The return value can be either an interval of type `Dom::Interval` or the empty set of type `DOM_SET`, if the interval is empty.
- Most mathematical operations are overloaded to work with intervals (such as `sin`). If f is a function of n real variables, its extension to intervals is defined to be $f(J_1, \ldots, J_n) := \{f(j_1, \ldots, j_n); j_i \in J_i\}$. The return value of such an operation is in most cases an interval, a union of intervals, a `Dom::ImageSet` or a set. For example, the sine of an interval `[a, b]` is the interval `{sin(x), x in [a, b]}` that contains all sine values of the given interval. In general, you should expect the return value to be an interval larger than strictly necessary. Also note that, when using the same interval twice in one formula, the uses are regarded as independent, so `interval1/interval1` does not return the interval `[1, 1]` as you might expect.

  The functions overloaded in this way are:

  - `_mult`, `_divide`, `_invert`, `_power`
  - `_plus`, `_negate`, `_subtract`
  - `abs`
  - `cos`, `arccos`, `cosh`, `arccosh`, `cot`, `arccot`, `coth`, `arccoth`, `csc`, `arccsc`, `csch`, `arccsch`, `sec`, `arcsec`, `sech`, `arcsech`, `sin`, `arcsin`, `sinh`, `arcsinh`, `tan`, `arctan`, `tanh`, `arctanh`
  - `dirac`, `heaviside`
  - `exp`, `ln`
  - `sign`
- Furthermore, an interval is a special type of set. This is reflected by `Dom::Interval` having the category `Cat::Set`. Among the methods inherited from `Cat::Set`, the following are especially important: `_intersect`, `_minus` and `_union`.
- An interval can be open or closed. If one border is given as a list with one element `[x]`, then this element `x` is taken as border and the interval will be created as closed at this side. If the interval should be closed at both sides, *one* list with the both borders as arguments can be given.

Entries:

`one`	the unit element; it equals the one-point interval $[1, 1]$.
`zero`	the zero element; it equals the one-point interval $[0, 0]$.

Mathematical Methods

Method `Im`: the imaginary part of an interval (this always equals zero)

`Im(`*`dom`* `interval)`

- This method returns the imaginary part of an interval (which is zero).
- This method overloads `Im`.

Method `Re`: the real part of an interval (this is the interval)

`Re(`*`dom`* `interval)`

- This method returns the real part of an interval (which is the interval).
- This method overloads `Re`.

Method `contains`: containing an element

`contains(`*`dom`* `interval,` *`any`* `element)`

- This method returns `TRUE` if `element` is an element of `interval`, `FALSE` if `element` is not an element of `interval`, and `UNKNOWN` if the property mechanism could not prove either statement.
- This method overloads `contains`.

Method `max`: maximum of an interval

`max(`*`dom`* `interval, ...)`

- This method returns the maximum of intervals. The return value is always an interval.
- This method overloads `max`.

Method `min`: minimum of an interval

```
min(dom interval, ...)
```

- This method returns the minimum of intervals. The return value is always an interval.
- This method overloads `min`.

Method `new`: create an interval

```
new(Type::Arithmetical left, Type::Arithmetical right)
new(Type::Arithmetical [left], Type::Arithmetical right)
new(Type::Arithmetical left, Type::Arithmetical [right])
new(Type::Arithmetical [left], Type::Arithmetical [right])
```

- This method creates a new interval with the borders `left` and `right`.

Access Methods

Method `borders`: the borders of an interval

```
borders(dom interval)
```

- This method returns a list with the borders of an interval.

Method `left`: the left border of an interval

```
left(dom interval)
```

- This method returns the left border of an interval.

Method `leftB`: the left border of an interval

```
leftB(dom interval)
```

- This method returns the left border of an interval. If the interval is left closed, then a list with the left border as element will be returned.

Method `isleftopen`: a left open interval

```
isleftopen(dom interval)
```

- This method returns `TRUE` if the interval is left open.

Method `isrightopen`: a right open interval

```
isrightopen(dom interval)
```

- This method returns `TRUE` if the interval is right open.

Method `iszero`: null interval

```
iszero(dom interval)
```

- This method returns `TRUE` if the given interval contains only the null, otherwise `FALSE`.
- This method overloads `iszero`.

Method `op`: the operands (borders) of an interval

```
op(dom interval)
```

- This method returns the both borders of an interval as given to create this interval, i.e., closed borders will be returned as a list with the border as operand.
- This method overloads `op`.

Method `subs`: substitution in intervals

```
subs(dom Interval, Type::Equation equation, ...)
```

- This method realizes substitution in intervals. The second and more arguments are the same as for the function `subs`.
- This method overloads `subs`.

Method `subsleft`: substitute left border

```
subsleft(dom interval, Type::Arithmetical left)
```

- This method substitutes the left border of `interval` by `left`. If the border will be given as list with one element, the border will be taken as closed.

Method `subsright`: substitute right border

```
subsright(dom interval, Type::Arithmetical right)
```

- This method substitutes the right border of `interval` by `right`. If the border will be given as list with one element, the border will be taken as closed.

Method `subsvals`: substitute both borders

`subsvals(`*`dom`* `interval, `*`Type::Arithmetical`* `left, `*`Type::Arithmetical`* `right)`

- This method substitutes both borders of `interval`. The call is the same as in `"subsleft"` and `"subsright"`.

Conversion Methods

Method `convert`: converting objects to intervals

`convert(`*`Any`* `object)`

- This method tries to convert `object` to an interval. Objects that can be converted are numbers, sets, properties, and arbitrary expressions.
- If the conversion fails, `FAIL` is returned.

Method `expr`: convert intervals to expressions

`expr(`*`dom`* `interval)`

- Converts an interval to an expression of type `_range`.
- This method overloads `expr`.

`expr(`*`dom`* `interval, `*`ident`* `x)`

- Returns a Boolean expression that is equivalent to `x in interval`.

Method `float`: convert to floating point interval

`float(`*`dom`* `interval)`

- This method maps the function `float` to the borders of the interval.
- This method overloads `float`.

Method `getElement`: one element of an interval

`getElement(`*`dom`* `interval)`

- This method returns one element of the given interval.

Method `simplify`: simplify intervals

`simplify(`*`dom`* `interval)`

- This method tries to simplify a given interval.
- This method overloads `simplify`.

Technical Methods

Method `emptycheck`: check intervals

`emptycheck(`*dom* `interval)`

- This method returns the given interval, if it is non empty, otherwise the empty set.

Method `equal`: comparison of intervals

`equal(`*dom* `interval, `*dom* `interval)`

- This method compares two given intervals and returns `TRUE`, if the given intervals are equal, otherwise `FALSE`.

Method `map`: apply functions to intervals

`map(`*dom* `interval, `*Type::Function* `function <`*Any* `argument, ...>)`

- This method maps the function `function` to the given interval. Additional arguments `argument, ...` are passed to the given function (see `map`). The return value is a `Dom::ImageSet` (which may simplify to a set or a `Dom::Interval`).
- This method overloads `map`.

Method `mapBorders`: apply functions to the borders of an interval

`mapBorders(`*dom* `interval, `*Type::Function* `function <`*Any* `argument, ...>)`

- This method maps `function` to the borders of `interval`. If additional arguments are given, `function` is called with these as second, third, etc. argument.

Method `print`: printing intervals

`print(`*dom* `interval)`

- This method returns a string used for displaying the interval.
- This method overloads `print`.

Method `random`: random interval

`random()`

- This method returns a random interval with numbers as borders.

Method `zip`: combine intervals

`zip(`*`dom`* `interval, `*`dom`* `interval, `*`Type::Function`* `function)`

- This method combines the two intervals with the given function borderwise.
- This method overloads `zip`.

Example 1. First create a closed interval between `0` and `1`.

```
>> A:= Dom::Interval([0], [1])
                                [0, 1]
```

Now another open interval between `-1` and `1`.

```
>> B:= Dom::Interval(-1, 1)
                               ]-1, 1[
```

Intervals can be handled like other objects.

```
>> A + B, A - B, A*B, A/B
  ]-1, 2[, ]-1, 2[, ]-1, 1[, ]0, infinity[ union ]-infinity, 0[
>> 2*A, 1 - A, (A - 1)^2
                     [0, 2], [0, 1], [0, 1]
```

Example 2. Standard functions are overloaded to work with intervals.

```
>> sin(B), float(sin(B))
        ]-sin(1), sin(1)[, ]-0.8414709848, 0.8414709848[
```

Example 3. The next examples shows some technical methods to access and manipulate intervals.

Get the borders and open/closed information about intervals.

```
>> A:= Dom::Interval([0], [1]):
   Dom::Interval::left(A), Dom::Interval::leftB(A)

                                0, [0]

>> Dom::Interval::isleftopen(A), Dom::Interval::subsleft(A, -
1)

                            FALSE, [-1, 1]
```

Super-Domain: `Dom::BaseDomain`

Changes:

- `Dom::Interval` was complete reorganized internally.
- With a new syntax closed and open intervals can be created.

`Dom::Matrix` – matrices

`Dom::Matrix(R)` creates a domain of matrices over the component ring R.

Domain:

- `Dom::Matrix(<R>)`

Parameters:

`R` — a ring, i.e., a domain of category `Cat::Rng`; default is `Dom::ExpressionField()`

Details:

- `Dom::Matrix(R)` creates domains of matrices over a component domain `R` of category `Cat::Rng` (a ring, possibly without unit).

  If the optional parameter `R` is not given, the domain `Dom::ExpressionField()` is used.

- A vector with n entries is either an $n \times 1$ matrix (a column vector), or a $1 \times n$ matrix (a row vector).

- Arithmetical operations with matrices can be performed by using the standard arithmetical operators of MuPAD.

  E.g., if `A` and `B` are two matrices defined by `Dom::Matrix(R)`, `A + B` computes the sum, and `A * B` computes the product of the two matrices, provided that the dimensions are correct.

  Similarly, `A^(-1)` or `1/A` computes the inverse of a square matrix `A` if it exists, and returns `FAIL` otherwise. See example 1.

- Many system functions have been overloaded for matrices, such as `map`, `subs`, `has`, `zip`, `conjugate` to compute the complex conjugate of a matrix, `norm` to compute matrix norms, or `exp` to compute the exponential of a matrix.

- Most of the functions in MuPAD's linear algebra package `linalg` work with matrices. For example, to compute the determinant of a square matrix `A`, call `linalg::det(A)`. The command `linalg::gaussJordan(A)` performs Gauss-Jordan elimination on `A` to transform `A` to its reduced row echelon form.

  See the documentation of `linalg` for a list of available functions of this package.

- The domain `Dom::Matrix(R)` represents matrices over `R` of arbitrary size, and it therefore does not have any algebraic structure (other than being a *set* of matrices).

  The domain `Dom::SquareMatrix(n, R)` represents the *ring* of $n \times n$ matrices over R. The domain `Dom::MatrixGroup(m, n, R)` represents the *Abelian group* of $m \times n$ matrices over R.

- We use the following notations for a matrix A (an element of `Dom::Matrix(R)`):

  - nrows(A) denotes the number of rows of A.
  - ncols(A) denotes the number of columns of A.
  - A *row index* is an integer in the range from 1 to nrows(A).
  - A *column index* is an integer in the range from 1 to ncols(A).

Creating Elements:

- `Dom::Matrix(R)(Array)`
- `Dom::Matrix(R)(List)`
- `Dom::Matrix(R)(ListOfRows)`
- `Dom::Matrix(R)(Matrix)`
- `Dom::Matrix(R)(m, n)`
- `Dom::Matrix(R)(m, n, ListOfRows)`
- `Dom::Matrix(R)(m, n, f)`

- `Dom::Matrix(R)(m, n, List, Diagonal)`
- `Dom::Matrix(R)(m, n, g, Diagonal)`
- `Dom::Matrix(R)(m, n, List, Banded)`
- `Dom::Matrix(R)(1, n, List)`
- `Dom::Matrix(R)(m, 1, List)`

Parameters:

`Array`	— a one- or two-dimensional array
`Matrix`	— a matrix, i.e., an element of a domain of category `Cat::Matrix`
`m, n`	— matrix dimension (positive integers)
`List`	— a list of matrix components
`ListOfRows`	— a list of at most `m` rows; each row given as a list of at most `n` matrix components
`f`	— a function or a functional expression with two parameters (the row and column index)
`g`	— a function or a functional expression with one parameter (the row index)

Options:

`Diagonal`	— create a diagonal matrix
`Banded`	— create a banded Toeplitz matrix

Categories:

`Cat::Matrix(R)`

Related Domains: `Dom::SquareMatrix`, `Dom::MatrixGroup`

Details:

- `Dom::Matrix(R)(Array)` and `Dom::Matrix(R)(Matrix)` create a new matrix with the dimension and the components of `Array` and `Matrix`, respectively.

  The components of `Array` or `Matrix` are converted into elements of the domain `R`. An error message is issued if one of these conversions fails.

- `Dom::Matrix(R)(List)` creates an $m \times 1$ column vector with components taken from the nonempty list, where m is the number of entries of `List`.

- `Dom::Matrix(R)(ListOfRows)` creates an $m \times n$ matrix with components taken from the nested list `ListOfRows`, where m is the number of inner lists of `ListOfRows`, and n is the maximal number of elements of an inner list. Each inner list corresponds to a row of the matrix. Both m and n must be nonzero.

If an inner list has less than n entries, then the remaining components in the corresponding row of the matrix are set to zero.

The entries of the inner lists are converted into elements of the domain `R`. An error message is issued if one of these conversions fails.

- The call `Dom::Matrix(R)(m, n)` returns the $m \times n$ zero matrix.

  Use the method `"identity"` to create the $n \times n$ identity matrix.

- `Dom::Matrix(R)(m, n, ListOfRows)` creates an $m \times n$ matrix with components taken from the list `ListOfRows`.

  If $m \geq 2$ and $n \geq 2$, then `ListOfRows` must consist of at most `m` inner lists, each having at most `n` entries. The inner lists correspond to the rows of the returned matrix.

  If an inner list has less than `n` entries, then the remaining components of the corresponding row of the matrix are set to zero. If there are less than `m` inner lists, then the remaining lower rows of the matrix are filled with zeroes.

- `Dom::Matrix(R)(m, n, f)` returns the matrix whose (i, j)th component is the value of the function call `f(i,j)`. The row index i ranges from 1 to m and the column index j from 1 to n.

  The function values are converted into elements of the domain `R`. An error message is issued if one of these conversions fails.

- `Dom::Matrix(R)(1, n, List)` returns the $1 \times n$ row vector with components taken from `List`. The list `List` must have at most `n` entries. If there are fewer entries, then the remaining vector components are set to zero.

  The entries of the list are converted into elements of the domain `R`. An error message is issued if one of these conversions fails.

- `Dom::Matrix(R)(m, 1, List)` returns the $m \times 1$ column vector with components taken from `List`. The list `List` must have at most `m` entries. If there are fewer entries, then the remaining vector components are set to zero.

  The entries of the list are converted into elements of the domain `R`. An error message is issued if one of these conversions fails.

Option <*`Diagonal`*>:

- With the option *`Diagonal`*, diagonal matrices can be created with diagonal elements taken from a list, or computed by a function or a functional expression.

- `Dom::Matrix(R)(m, n, List, `*`Diagonal`*`)` creates the $m \times n$ diagonal matrix whose diagonal elements are the entries of `List`.

`List` must have at most $\min(m, n)$ entries. If it has fewer elements, the remaining diagonal elements are set to zero.

The entries of `List` are converted into elements of the domain `R`. An error message is issued if one of these conversions fails.

- `Dom::Matrix(R)(m, n, g, Diagonal)` returns the matrix whose ith diagonal element is `g(i)`, where the index i runs from 1 to $\min(m, n)$.

  The function values are converted into elements of the domain `R`. An error message is issued if one of these conversions fails.

Option <*Banded*>:

- `Dom::Matrix(R)(m, n, List, Banded)` creates an $m \times n$ banded Toeplitz matrix with the elements of `List` as entries. The number of entries of `List` must be odd, say $2h + 1$, and must not exceed `n`. The resulting matrix has bandwidth at most $2h + 1$.

  A Toeplitz matrix is a matrix where the elements of each band are identical. See also example 7.

  All elements of the main diagonal of the created matrix are initialized with the middle element of `List`. All elements of the ith subdiagonal are initialized with the $(h + 1 - i)$th element of `List`. All elements of the ith superdiagonal are initialized with the $(h + 1 + i)$th element of `List`. All entries on the remaining sub- and superdiagonals are set to zero.

  The entries of `List` are converted into elements of the domain `R`. An error message is issued if one of these conversions fails.

Entries:

`isSparse` is always `FALSE`, as elements of `Dom::Matrix(R)` use a dense representation of their matrix components.

`randomDimen` is set to `[10,10]`. See the method `"random"` below for details.

Mathematical Methods

Method `_divide`: divides matrices

`_divide(`*`dom`* `A, `*`dom`* `B)`

- This method computes the product $A \cdot B^{-1}$. The matrix `B` must be nonsingular, otherwise `FAIL` is returned.
- An error message is issued if the dimensions of `A` and `B` do not match.

- This method only exists if `R` is an integral domain, i.e., a domain of category `Cat::IntegralDomain`.
- This method overloads the function `_divide` for matrices, i.e., one may use it in the form `A / B`, or in functional notation: `_divide(A, B)`.

Method `_invert`: computes the inverse of a matrix

```
_invert(dom A)
```

- This method computes the inverse of the matrix `A`. If `A` is singular, `FAIL` is returned.
- If the component ring `R` is the domain `Dom::Float`, a floating-point approximation of the inverse matrix is computed by the function `numeric::inverse`.
- This method only exists if `R` is a domain of category `Cat::IntegralDomain`.
- This method overloads the function `_invert` for matrices, i.e., one may use it in the form `1/A` or `A^(-1)`, or in functional notation: `_invert(A)`.

Method `_mult`: multiplies matrices by matrices, vectors and scalars

```
_mult(dom x, any y)
```

- If `y` is a matrix of the same domain type as `x`, the matrix product $x * y$ is computed. An error message is issued if the dimensions of the matrices do not match.
- If `y` is of the domain type `R` or can be converted into such an element, the corresponding scalar multiplication is computed.

  Otherwise, `y` is converted into a matrix of the domain type of `x`. If this conversion fails, then this method calls the method `"_mult"` of the domain of `y` giving all arguments in the same order.

```
_mult(any x, dom y)
```

- If `x` is a matrix of the same domain type as `y`, then the matrix product $x * y$ is computed. An error message is issued if the dimensions of the matrices do not match.
- If `x` is of the domain type `R` or can be converted into such an element, the corresponding scalar multiplication is computed.

  Otherwise, `x` is converted into a matrix of the domain type of `y`. If this conversion fails, then `FAIL` is returned.
- This method handles more than two arguments by calling itself recursively with the first half of all arguments and the last half of all arguments. Then the product of these two results is computed with the system function `_mult`.

- This method overloads the function `_mult` for matrices, i.e., one may use it in the form `x * y`, or in functional notation: `_mult(x, y)`.

Method `_negate`: negates a matrix

```
_negate(dom A)
```

- The matrix $-A$ is returned.
- This method overloads the function `_negate` for matrices, i.e., one may use it in the form `-A`, or in functional notation: `_negate(A)`.

Method `_plus`: adds matrices

```
_plus(matrix A, matrix B, ...)
```

- Returns the matrix sum $A + B + \cdots$.
  An error message is issued if the given matrices do not have the same dimensions.
- The arguments `A, B, ...` are converted into matrices of the domain type `Dom::Matrix(R)`. `FAIL` is returned if one of these conversions fails.
- This method overloads the function `_plus` for matrices, i.e., one may use it in the form `A + B`, or in functional notation: `_plus(A, B)`.

Method `_power`: the integer power of a matrix

```
_power(dom A, integer n)
```

- This method computes A^n. If `A` is not square, `FAIL` is returned.
- If the power `n` is a negative integer then `A` must be nonsingular and `R` must be a domain of category `Cat::IntegralDomain`. Otherwise `FAIL` is returned.
- If `n` is zero and the component ring `R` is a ring with no unit (i.e., of category `Cat::Rng`, but not of category `Cat::Ring`), `FAIL` is returned.
- This method overloads the function `_power` for matrices, i.e., one may use it in the form `A^n`, or in functional notation: `_power(A, n)`.

Method `conjugate`: the complex conjugate of a matrix

```
conjugate(dom A)
```

- The complex conjugate matrix of `A` is the matrix obtained by computing the complex conjugate of each component of `A`.
- This method only exists if `R` implements the method `"conjugate"`, which computes the complex conjugate of an element of the domain `R`.
- This method overloads the function `conjugate` for matrices, i.e., one may use it in the form `conjugate(A)`.

Method `diff`: differentiation of matrix components

```
diff(dom A, ...)
```

- This method differentiates each component of the matrix `A` using the method `"diff"` of the component ring `R`.
  Additional arguments are passed to the method `"diff"` of the domain `R`. See the system function `diff` for details.
- This method only exists if `R` implements the method `"diff"`.
- This method overloads the function `diff` for matrices, i.e., one may use it in the form `diff(A, ...)`.

Method `equal`: equality test of matrices

```
equal(dom A, dom B)
```

- This method tests if the two matrices `A` and `B` are equal and returns `TRUE`, `FALSE`, or `UNKNOWN`, respectively.
- Note that if `R` has the axiom `Ax::systemRep` then `normal` is used to simplify the components of `A` and `B` before testing their equality.

Method `exp`: the exponential of a matrix

```
exp(dom A<, R t>)
```

- This method computes the matrix exponential of the $n \times n$ matrix `A`, defined by $I_n + At + \frac{1}{2}(At)^2 + \ldots$, where I_n is the $n \times n$ identity matrix. The default value of `t` is 1.
- If `A` is not square, an error message is issued.
- This method uses the function `numeric::expMatrix` for a floating-point approximation of the exponential of `A` if `A` is defined over the domain `Dom::Float` and if $t = 1$.
- If some eigenvalues of `A` do not exist in `R` or cannot be computed, then `FAIL` is returned.

- In the symbolic case the function `linalg::jordanForm` is called, which may not be able to compute the Jordan form of `A`. In this case `FAIL` is returned. Increasing the level of information (see `setuserinfo`) can yield useful information.
- This method only exists if `R` is a domain of category `Cat::Field`.
- This method overloads the function `exp` for matrices, i.e., one may use it in the form `exp(A, ...)`.

Method `expand`: expand matrix components

`expand(`*`dom`* `A)`

- This method applies the function `expand` to each component of the matrix `A`.
- This method only exists if `R` implements the method `"expand"`, or if `R` has the axiom `Ax::systemRep` (in this case, the system function `expand` is used).
- This method overloads the function `expand` for matrices, i.e., one may use it in the form `expand(A)`.

Method `factor`: scalar-matrix factorization

`factor(`*`dom`* `A)`

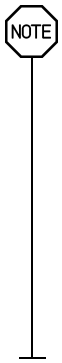
- This method factorizes `A` into the form $A = s \cdot B$, where s is a scalar of the component ring `R`.
  The result is a factored object, i.e., an element of the domain `Factored`. It has the factorization type `"unknown"`.
- The factor s is the gcd of all components of the matrix `A`. Hence, this method only exists if `R` is of category `Cat::GcdDomain`.
- This method overloads the function `factor` for matrices, i.e., one may use it in the form `factor(A)`.

Method `float`: floating-point approximation of the matrix components

`float(`*`dom`* `A)`

- This method maps the function `float` to the matrix components of `A`, i.e., it computes a floating-point approximation of the matrix components.
- This method only exists if `R` implements the method `"float"`.

- Usually the floating-point approximations are not elements of R! For example, `Dom::Integer` implements such a method, but the floating-point approximation of an integer cannot be re-converted into an integer.
  This method checks whether the resulting matrix can be converted into the domain type of A only if `testargs` returns TRUE (e.g., if one calls this method from the interactive level of MuPAD).
  Otherwise, one has to take care that the matrix returned is compatible to its component ring.



Method `gaussElim`: Gaussian elimination

`gaussElim(`*dom* `A)`

- This method performs the Gaussian elimination on A and reduces A to an upper row echelon form T.
  It returns a list containing the matrix T, the rank and determinant of A, and the set of characteristic column indices of T (in this order).
- If the matrix is not square, i.e., the determinant of A is not defined, then the third entry of the list returned is the value `FAIL`.
- This method only exists if the component ring R is an integral domain, i.e., a domain of category `Cat::IntegralDomain`.
- If R has the method `"pivotSize"`, then the pivot element of smallest size is chosen at every pivoting step, whereby `pivotSize` must return a positive integer representing the "size" of an element.
  If no such method is defined, Gaussian elimination without a pivot strategy is applied to A.
- If R has the axiom `Ax::efficientOperation("_invert")` and is of category `Cat::Field`, then ordinary Gaussian elimination is used. Otherwise, fraction-free elimination is performed on A.
- If R implements the method `"normal"`, it is used to simplify subsequent computations of the Gaussian elimination process.
  Note that if R does not implement the method `"normal"`, but the elements of R are represented by kernel domains, i.e., R has the axiom `Ax::systemRep`, the system function `normal` is used instead.

Method `identity`: identity matrix

`identity(`*positive integer* `n)`

- This method returns the $n \times n$ identity matrix.
- This method only exists if the component ring R is of category `Cat::Ring`, i.e., a ring with unit.

Method `iszero`: test for zero matrices

`iszero(`*`dom`* `A)`

- This method checks whether `A` is a zero matrix.
- Note that there may exist more than one representation of the zero matrix of a given dimension if `R` does not have `Ax::canonicalRep`.
- If `R` implements the method `"normal"`, it is used to simplify the components of `A` for the zero-test.
  Note that if `R` does not implement such a method, but the elements of `R` are represented by kernel domains, i.e., `R` has the axiom `Ax::systemRep`, the system function `normal` is used instead.
- This method overloads the function `iszero` for matrices, i.e., one may use it in the form `iszero(A)`.

Method `matdim`: matrix dimension

`matdim(`*`dom`* `A)`

- This method returns the number of rows and columns of the matrix `A` as a list of two positive integers.

Method `norm`: norm of matrices and vectors

`norm(`*`dom`* `A <,` *`Infinity`*`>)`

- Computes the infinity norm of the matrix `A`, which is the maximum row sum (the row sum is the sum of norms of each component in a row).
  If the domain `R` does not implement the methods `"max"` and `"norm"`, `FAIL` is returned.

`norm(`*`dom`* `v <,` *`Infinity`*`>)`

- For a vector `v` the maximum norm of all elements is returned.
  If the domain `R` does not implement the methods `"max"` and `"norm"`, `FAIL` is returned.

`norm(`*`dom`* `A,` *`Frobenius`*`)`

- Computes the Frobenius norm of `A`, which is the square root of the sum of the squares of the norms of each component.
  If the result is no longer an element of the domain `R`, or if `R` does not implement the method `"norm"`, `FAIL` is returned.

`norm(`*`dom`* `A, 1)`

- Computes the 1-norm of the matrix `A`, which is the maximum sum of the norms of the elements of each column. If `R` does not implement the methods `"max"` and `"norm"`, `FAIL` is returned.

`norm(`*dom* `v, `*positive integer* `k)`

- Computes the k-norm of the vector `v`, which is defined to be the kth root of the sum of the norms of the elements of `v` raised to the kth power.
  `FAIL` is returned if the result is no longer an element of the domain `R`. For $k = 2$, the function `linalg::scalarProduct` is used to compute the 2-norm of `v`.
  If `R` does not implement the method `"norm"`, `FAIL` is returned.
- This method overloads the function `norm` for matrices, i.e., one may use it in the form `norm(A<, k>)`, where `k` is either *`Infinity`*, *`Frobenius`*, or a positive integer. The default value of `k` is *`Infinity`*.

Method `normal`: simplification of matrix components

`normal(`*dom* `A)`

- The method `"normal"` of `R` is applied to the components of `A`.
- If `R` does not implement the method `"normal"`, but the elements of `R` are represented by kernel domains, i.e., `R` has the axiom `Ax::systemRep`, then the system function `normal` is applied to the components of `A`. Otherwise `normal(A)` returns `A` without any changes.
- This method overloads the function `normal` for matrices, i.e., one may use it in the form `normal(A)`.

Method `nonZeros`: number of non-zero components of a matrix

`nonZeros(`*dom* `A)`

- This method returns the number of components of `A` for which the method `"iszero"` of the component ring `R` returns `FALSE`.

Method `random`: random matrix generation

`random()`

- This method returns a random matrix. It uses the method `"random"` of the component ring `R` to randomly generate the components of the matrix.
- This method only exists if `R` implements the method `"random"`.
- The dimension of the matrix is also chosen randomly, but it is limited by the values given in `"randomDimen"` (see "Entries" above).
- To change the value of the entry `"randomDimen"` for a domain `MatR` created with `Dom::Matrix`, one must first unprotect the domain `Dom` (see `unprotect` for details).

Method `tr`: trace of a square matrix

`tr(`*`dom`* `A)`

- This method computes the trace of the square matrix `A`, which is defined to be the sum of its diagonal entries.
- If `A` is not square, then an error message is issued.

Method `transpose`: transpose of a matrix

`transpose(`*`dom`* `A)`

- This method returns the transpose matrix A^t of `A`.

Access Methods

Method `_concat`: horizontal concatenation of matrices

`_concat(`*`dom`* `A, `*`dom`* `B, ...)`

- This method appends the matrices `B, ...` to the right side of the matrix `A`.
- An error message is issued if the given matrices do not have the same number of rows.
- This method overloads the function `_concat` for matrices, i.e., one may use it in the form `A . B . ...`, or in functional notation: `_concat(A, B, ...)`.

Method `_index`: matrix indexing

`_index(`*`dom`* `A, `*`row index`* `i, `*`column index`* `j)`

- This method returns the (i, j)th entry of the matrix `A`.

`_index(`*`dom`* `A, `*`row-range`* `r1..r2, `*`column-range`* `c1..c2)`

- This method returns the submatrix of `A` created by the rows of `A` with indices from `r1` to `r2` and the columns of `A` with indices from `c1` to `c2`.

`_index(`*`dom`* `v, `*`index`* `i)`

- This method returns the ith entry of the vector `v`.
- An error message is issued if `v` is not a vector.

`_index(`*`dom`* `v, `*`index-range`* `i1..i2)`

- This method returns the subvector of `v`, formed by the entries with index `i1` to `i2`. See also the method `"op"`.

- An error message is issued if `v` is not a vector.
- This method overloads the function `_index` for matrices, i.e., one may use it in the form `A[i, j]`, `A[r1..r2, c1..c2]`, `v[i]` and `v[i1..i2]`, respectively, or in functional notation: `_index(A, ...)`.

Method `concatMatrix`: horizontal concatenation of matrices

```
concatMatrix(dom A, dom B, ...)
```

- This method is identical to the method `"_concat"`.

Method `col`: extracting a column

```
col(dom A, column index c)
```

- This method extracts the column with index `c` of the matrix `A` and returns it as a column vector, i.e., as an element of type `Dom::Matrix(R)`.
- An error message is issued if `c` is less than one or greater than the number of columns of `A`.

Method `delCol`: deleting a column

```
delCol(dom A, column index c)
```

- This method returns the matrix obtained by deleting the column with index `c` of the matrix `A`.
- `NIL` is returned if `A` consists of only one column.
- An error message is issued if `c` is less than one or greater than the number of columns of `A`.

Method `delRow`: deleting a row

```
delRow(dom A, row index r)
```

- This method returns the matrix obtained by deleting the row with index `r` of the matrix `A`.
- `NIL` is returned if `A` consists of only one row.
- An error message is issued if `r` is less than one or greater than the number of rows of `A`.

Method `evalp`: evaluating matrices of polynomials at a certain point

```
evalp(dom A, equation x = a, ...)
```

- This method evaluates the polynomial components of `A` at the point `x = a`. See the system function `evalp` for details.

  The matrix returned is of the domain `Dom::Matrix(R::coeffRing)` if the evaluation of all components leads to an element of the coefficient ring of the polynomial domain. Otherwise the matrix returned is of the domain of `A`.
- This method is only defined if `R` is a polynomial ring of category `Cat::Polynomial`.
- This method overloads the function `evalp` for matrices, i.e., one may use it in the form `evalp(A, x = a)`.

Method `length`: length of a matrix

```
length(dom A)
```

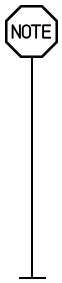
- This method returns the length of the matrix `A`, which is the length of the array holding the components of `A`. See the system function `length` for details.
- This method overloads the function `length` for matrices, i.e., one may use it in the form `length(A)`.

Method `map`: apply a function to matrix components

```
map(dom A, function func<, any expr, ...>)
```

- This method maps the function `func` to the components of the matrix `A`, with the additional function parameters `expr, ...` passed to `func`, if given.

  See the system function `map` for details.
- Note that the function values are converted into elements of the domain `R` only if `testargs` returns `TRUE` (e.g., if one calls this method from the interactive level of MuPAD).
  If `testargs` returns `FALSE`, then one must guarantee that the function calls return elements of the domain type `R`, otherwise the resulting matrix, which is of domain type `Dom::Matrix(R)`, would have components which are not elements of the domain `R`!



- This method overloads the function `map` for matrices, i.e., one may use it in the form `map(A, func, ...)`.

Method `nops`: number of components of a matrix

`nops(`*`dom`* `A)`

- This method returns the number of components of a matrix, which is $m \cdot n$ for an $m \times n$ matrix `A`.
- This method overloads the function `nops` for matrices, i.e., one may use it in the form `nops(A)`.

Method `op`: components of a matrix

`op(`*`dom`* `A,` *`positive integer`* `i)`

- This method returns the ith component of the matrix `A`, where the components are numbered starting at row one from left to right and up to down.

`op(`*`dom`* `A)`

- This method returns an expression sequence of all components of `A`.
- See also the method `"_index"`.
- This method overloads the function `op` for matrices, i.e., one may use it in the form `op(A, i)` and `op(A)`, respectively.

Method `row`: extracting a row

`row(`*`dom`* `A,` *`row index`* `r)`

- This method extracts the row with index `r` of the matrix `A` and returns it as a row vector, i.e., as an element of type `Dom::Matrix(R)`.
- An error message is issued if `r` is less than one or greater than the number of rows of `A`.

Method `setCol`: replacing a column

`setCol(`*`dom`* `A,` *`column index`* `c,` *`dom`* `v)`

- This method replaces the column with index `c` of the matrix `A` by the column vector `v`. The vector `v` must have nrows(A) elements.
- An error message is issued if `c` is less than one or greater than the number of rows of `A`.

Method `setRow`: replacing a row

`setRow(`*`dom`* `A, `*`row index`* `r, `*`dom`* `v)`

- This method replaces the row with index `r` of the matrix `A` by the row vector `v`. The vector `v` must have ncols(A) elements.
- An error message is issued if `r` is less than one or greater than the number of rows of `A`.

Method `stackMatrix`: vertical concatenation of matrices

`stackMatrix(`*`dom`* `A, `*`dom`* `B, ...)`

- This method stacks the matrix `A` on the top of the matrix `B`. If further arguments are given, then the result is stacked on the top of the third matrix, and so on.
- An error message is issued if the given matrices do not have the same number of columns.

Method `subs`: substitution of matrix components

`subs(`*`dom`* `A, ...)`

- This method maps the function `subs` with additionally given parameters to the components of the matrix `A`. See the system function `subs` for details.
- Note that the function values are converted into elements of the domain `R` only if `testargs` returns `TRUE` (e.g., if one calls this method from the interactive level of MuPAD).
  If `testargs` returns `FALSE`, then one must guarantee that the function calls return elements of the domain type `R`, otherwise the resulting matrix, which is of domain type `Dom::Matrix(R)`, would have components which are not elements of the domain `R`!

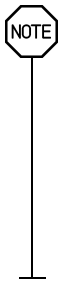
- This method overloads the function `subs` for matrices, i.e., one may use it in the form `subs(A, ...)`.

Method `subsex`: extended substitution of matrix components

`subsex(`*`dom`* `A, ...)`

- This method maps the function `subsex` with additionally given parameters to the components of the matrix `A`. See the system function `subsex` for details.

- Note that the results of the substitutions are converted into elements of the domain `R` only if `testargs` returns `TRUE` (e.g., if one calls this method from the interactive level of MuPAD). If `testargs` returns `FALSE`, then one must guarantee that the results of the substitutions are of the domain type `R`, otherwise the resulting matrix, which is of domain type `Dom::Matrix(R)`, would have components which are not elements of the domain `R`!

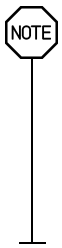


- This method overloads the function `subsex` for matrices, i.e., one may use it in the form `subsex(A, ...)`.

Method `subsop`: operand substitution of matrix components

`subsop(`*`dom`* `A, `*`equation`* `i = x, ...)`

- This method replaces the *i*th component of the matrix `A` by `x`.
- Note that `x` is converted into the domain `R` only if `testargs` returns `TRUE` (e.g., if one calls this method from the interactive level of MuPAD).
  If `testargs` returns `FALSE`, then `x` must be an element of `R`, otherwise the resulting matrix, which is of domain type `Dom::Matrix(R)`, would have components which are not elements of the domain `R`!



- See also the method `"set_index"`.
- This method overloads the function `subsop` for matrices, i.e., one may use it in the form `subsop(A, ...)`.

Method `swapCol`: swapping matrix columns

`swapCol(`*`dom`* `A, `*`column indices`* `c1, c2)`

- This method returns the matrix which results from swapping the column with index `c1` with the column with index `c2` of the matrix `A`.
- An error message is issued if one of the column indices is less than one or greater than the number of columns of `A`.

`swapCol(`*`dom`* `A, `*`column indices`* `c1, c2, `*`row range`* `r1..r2)`

- This method swaps the column with index `c1` and the column with index `c2` of `A`, but by taking only those column components which lie in the rows with indices `r1` to `r2`.
- An error message is issued if one of the column indices is less than one or greater than the number of columns of `A`, or if one of the row indices is less than one or greater than the number of rows of `A`.

Method `swapRow`: swapping matrix rows

`swapRow(`*dom* `A, `*row indices* `r1, r2)`

- This method returns the matrix which results from swapping the row with index `r1` with row with index `r2` of the matrix `A`.
- An error message is issued if one of the row indices is less than one or greater than the number of rows of `A`.

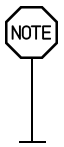
`swapRow(`*dom* `A, `*row indices* `r1, r2, `*column range* `c1..c2)`

- This method swaps the row with index `r1` and the row with index `r2` of `A`, but by taking only those row components which lie in the columns with indices `c1` to `c2`.
- An error message is issued if one of the row indices is less than one or greater than the number of rows of `A`, or if one of the column indices is less than one or greater than the number of columns of `A`.

Method `set_index`: setting matrix components

`set_index(`*dom* `A, `*row index* `i, `*column index* `j, `*any* `x)`

- Replaces the (i, j)th component of the matrix `A` by `x`.
- Note that `x` is converted into an element of the domain `R` only if `testargs` returns `TRUE` (e.g., if one calls this method from the interactive level of MuPAD).
  Otherwise one has to take care that `x` is of domain type `R`.



- See also the method `"subsop"`.

`set_index(`*dom* `v, `*index* `i, `*any* `x)`

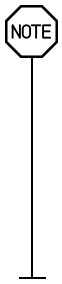
- Replaces the ith entry of the vector `v` by `x`.
- This method overloads the function `set_index` for matrices, i.e., one may use it in the form `A[i, j] := x` and `v[i] := x`, respectively, or in functional notation: `A := set_index(A, i, j, x)` or `v := set_index(v, i, x)`.

Method `zip`: combine matrices component-wise

`zip(`*dom* `A, B, `*function* `func<, `*any* `expr, ...>)`

- This method combines the matrices `A` and `B` component-wise, where the function `func(a, b<, expr, ...>)` is applied to each pair (A_{ij}, B_{ij}) (for all i and j).
- The row number of the matrix returned is the minimum of the row numbers of `A` and `B`, and its column number is the minimum of the column numbers of `A` and `B`.

- Note that the function values are converted into elements of the domain `R` only if `testargs` returns `TRUE` (e.g., if one calls this method from the interactive level of MuPAD).
  If `testargs` returns `FALSE`, then one must guarantee that the function calls return elements of the domain type `R`, otherwise the resulting matrix, which is of domain type `Dom::Matrix(R)`, would have components which are not elements of the domain `R`!



- This method overloads the function `zip` for matrices, i.e., one may use it in the form `zip(A, B, ...)`.

Conversion Methods

Method `convert`: conversion to a matrix

```
convert(any x)
```

- This method tries to convert `x` into a matrix of type `Dom::Matrix(R)`.
- `FAIL` is returned if the conversion fails.
- `x` may either be an array, a matrix, or a list (of sublists, see the parameter `ListOfRows` in "Creating Elements" above). Their entries must then be convertible into elements of the domain `R`.

Method `convert_to`: matrix conversion

```
convert_to(dom A, any T)
```

- This method tries to convert the matrix `A` into an element of domain type `T`. `FAIL` is returned if the conversion fails.
- `T` may either be `DOM_ARRAY`, `DOM_LIST`, or a domain constructed by `Dom::Matrix` or `Dom::SquareMatrix`. The elements of `A` must be convertible into elements of the domain `R`.
- Use the function `expr` to convert `A` into an object of a kernel domain (see below).

Method `create`: defining matrices without component conversions

```
create(any x, ...)
```

- This method creates a new matrix assuming that the components are of domain type `R`.
  See "Creating Elements" above for a complete description of the parameters, with one exception: one *cannot* use this method to create a matrix from a function or a functional expression.

- This method works more efficient than if one creates matrices by calling the method `"new"` of the domain, because it avoids any conversion of the components. One must guarantee that the components have the correct domain type, otherwise run-time errors can be caused.

Method `expr`: matrix conversion into an object of a kernel domain

`expr(`*`dom`* `A)`

- This method converts `A` into an array, i.e., an object of type `DOM_ARRAY`, and applies the function `expr` to each component of `A`.
- The result is an array representing the matrix `A` where each entry is an object of a kernel domain.
- This method overloads the function `expr` for matrices, i.e., one may use it in the form `expr(A)`.

Method `expr2text`: matrix conversion to a string

`expr2text(`*`dom`* `A)`

- This method converts `A` into a string `s` such that the evaluation of the function call `text2expr(s)` gives the matrix `A`.
- This method overloads the function `expr2text` for matrices, i.e., one may use it in the form `expr2text(A)`.

Method `TeX`: TeX formatting of a matrix

`TeX(`*`dom`* `A)`

- This method returns a TEX-formatted string for the matrix `A` in form of a TEX `array` environment.
- The method `"TeX"` of the component ring `R` is used to get the TEX-representation of each component of `A`.
- This method is used by the function `generate::TeX`.

Technical Methods

Method `assignElements`: multiple assignment to matrices

`assignElements(`*`dom`* `A, ...)`

- This method performs multiple assignments to components of the matrix `A`.

  See the system function `assignElements` for details.

- The assigned components must have the domain type `R`, an implicit conversion of the components into elements of domain type `R` is not performed.
- This method overloads the function `assignElements` for matrices, i.e., one may use it in the form `assignElements(A, ...)`.

Method `mkDense`: conversion of a matrix to an array

`mkDense(`*`array`* `Array)`

- This method converts each operand of `Array` into an element of the component ring `R`. The result is either `FAIL` if one of these conversions is not possible, or the list `[r, c, Array]`, where the positive integers `r` and `c` give the dimension of `Array`.

`mkDense(`*`list`* `List)`

- This method tries to convert the list `List` into an array `a`. The result is either `FAIL` if this is not possible, or the list `[r, c, a]`, where the positive integers `r` and `c` give the dimension of `a`. See the parameters `List` and `ListOfRows` in "Creating Elements" above for admissible formats of `List`.

  The array `a` has dimension one if `r` or `c` is equal to one. The entries of `a` have been converted into elements of the domain `R`.

`mkDense(`*`positive integers`* `r, c,` *`list`* `List)`

- This method tries to convert the list `List` into an array `a` of the dimension `r` times `c`.

  The result is either `FAIL` if this is not possible, or the list `[r, c, a]`.

  The array `a` has dimension one if `r` or `c` is equal to one. The entries of `a` have been converted into elements of the domain `R`.

Method `print`: printing matrices

`print(`*`dom`* `A)`

- This method returns the array holding the components of `A`. Thus, matrices are printed like arrays.

Method `unapply`: create a procedure from a matrix

`unapply(`*`dom`* `A<,` *`identifier`* `x, ...>)`

- This method interprets the components of `A` as functions in `x, ...`, and returns a procedure representing this matrix function. See `fp::unapply` for details.

- This method overloads the function `fp::unapply` for matrices, i.e., one may use it in the form `fp::unapply(A)`.

Example 1. First we create the domain of matrices over the field of rational numbers:

```
>> MatQ := Dom::Matrix(Dom::Rational)

                    Dom::Matrix(Dom::Rational)
```

We assigned this domain to the identifier `MatQ`. Next we define the 2×2 matrix

$$\begin{pmatrix} 1 & 5 \\ 2 & 3 \end{pmatrix}$$

by a list of two rows, where each row is a list of two elements:

```
>> A := MatQ([[1, 5], [2, 3]])

                              +-      -+
                              |  1, 5  |
                              |        |
                              |  2, 3  |
                              +-      -+
```

In the same way we define the following 2×3 matrix:

```
>> B := MatQ([[-1, 5/2, 3], [1/3, 0, 2/5]])

                          +-                -+
                          |   -1, 5/2,  3    |
                          |                  |
                          |  1/3,  0,  2/5   |
                          +-                -+
```

and perform matrix arithmetic using the standard arithmetical operators of MuPAD, e.g., the matrix product $A \cdot B$, the 4th power of A as well as the scalar multiplication of A times $\frac{1}{3}$:

```
>> A * B, A ^ 4, 1/3 * A

      +-                 -+  +-           -+  +-           -+
      |  2/3, 5/2,   5    |  |  281, 600  |  |  1/3, 5/3  |
      |                   |, |            |, |            |
      |   -1,  5,  36/5   |  |  240, 521  |  |  2/3,  1   |
      +-                 -+  +-           -+  +-           -+
```

The matrices A and B have different dimensions, and therefore the sum of A and B is not defined. MuPAD issues an error message:

```
>> A + B

 Error: dimensions don't match [(Dom::Matrix(Dom::Rational))\
 ::_plus]
```

To compute the inverse of A, just enter:

```
>> 1/A
```

$$\begin{pmatrix} -3/7 & 5/7 \\ 2/7 & -1/7 \end{pmatrix}$$

If a matrix is not invertible, `FAIL` is the result of this operation. For example, the matrix:

```
>> C := matrix(2, 2, [[2]])
```

$$\begin{pmatrix} 2 & 0 \\ 0 & 0 \end{pmatrix}$$

is not invertible, hence:

```
>> C^(-1)
```

```
FAIL
```

Example 2. We create the domain of matrices over the reals:

```
>> MatR := Dom::Matrix(Dom::Real)
```

```
Dom::Matrix(Dom::Real)
```

Beside standard matrix arithmetic, the library `linalg` offers a lot of functions dealing with matrices. For example, if one wants to compute the rank of a matrix, use `linalg::rank`:

```
>> A := MatR([[1, 2], [2, 4]])
```

$$\begin{pmatrix} 1 & 2 \\ 2 & 4 \end{pmatrix}$$

```
>> linalg::rank(A)
```

```
                                  1
```

Use `linalg::eigenvectors` to compute eigenvalues and eigenvectors of the matrix A:

```
>> linalg::eigenvectors(A)
```

```
  -- --       -- +-    -+ -- --  --       -- +-     -+ -- -
- --
  |  |       |  |  -2  |  |  |  |       |  |  1/2  |  |  |  |
  |  |  0, 1, |  |      |  |  |, |  5, 1, |  |       |  |  |  |
  |  |       |  |   1  |  |  |  |  |       |  |   1   |  |  |  |
  -- --       -- +-    -+ -- --  --       -- +-     -+ -- -
- --
```

Try `info(linalg)` for a list of available functions, or enter `help(linalg)` for details about the library `linalg`.

Some of the functions in the `linalg` package simply serve as "interface" functions for methods of a matrix domain described above. For example, `linalg::transpose` uses the method `"transpose"` to get the transposed matrix. The function `linalg::gaussElim` applies Gaussian elimination to a matrix, such as:

```
>> linalg::gaussElim(A)
```

```
                          +-      -+
                          |  1, 2  |
                          |        |
                          |  0, 0  |
                          +-      -+
```

The computation is performed by the method `"gaussElim"` as described above. Such functions of the `linalg` packages, in contrast to the corresponding methods of the domain `Dom::Matrix(R)`, check their incoming parameters, and some of them offer extended functionalities.

Example 3. In this example, we use the default matrix domain which is created by `Dom::Matrix()`. This domain represents matrices whose components can be arbitrary arithmetical expressions (i.e., the component ring is the domain `Dom::ExpressionField()`).

This domain is already known to MuPAD by the name `matrix`:

```
>> A := matrix(
     [[1, 2, 3, 4], [2, 0, 4, 1], [-1, 0, 5, 2]]
   )
```

```
+-              -+
|   1, 2, 3, 4   |
|                |
|   2, 0, 4, 1   |
|                |
|  -1, 0, 5, 2   |
+-              -+
```

```
>> domtype(A)
```

```
Dom::Matrix()
```

Matrix components can be extracted by the index operator `[]`:

```
>> A[2, 1] * A[1, 2] - A[3, 1] * A[1, 3]
```

```
7
```

If one of the indices is not in its valid range, an error message is issued. Assignments to matrix components are performed similarly:

```
>> delete a:
   A[1, 2] := a^2: A
```

```
+-               -+
|        2        |
|   1, a , 3, 4   |
|                 |
|   2,  0, 4, 1   |
|                 |
|  -1,  0, 5, 2   |
+-               -+
```

Beside the usual indexing of matrix components, it is also possible to extract submatrices from a given matrix. The following call creates the submatrix of A which consists of the rows 2 to 3 and columns 1 to 3 of A:

```
>> A[2..3, 1..3]
```

```
+-           -+
|   2, 0, 4   |
|             |
|  -1, 0, 5   |
+-           -+
```

The index operator does not allow to insert submatrices into a given matrix. This is implemented by the function `linalg::substitute`.

Example 4. In the following examples, we demonstrate the different ways of creating matrices. We work with matrices defined over the field $\mathbb{Z}_{19}$, i.e., the field of integers modulo 19. This component ring can be created with the domain constructor `Dom::IntegerMod`.

We start by giving a list of rows, where each row is a list of row entries:

```
>> MatZ19 := Dom::Matrix(Dom::IntegerMod(19)):
   MatZ19([[1, 2], [2]])
```

```
                  +-                     -+
                  |  1 mod 19, 2 mod 19   |
                  |                       |
                  |  2 mod 19, 0 mod 19   |
                  +-                     -+
```

The elements of the two inner lists, the row entries, were converted into elements of the domain `Dom::IntegerMod(19)`.

The number of rows is the number of sublists of the argument, i.e., $m = 2$. The number of columns is determined by the length of the inner list with the most entries, which is the first inner list with two entries. Missing entries in the other inner lists are treated as zero components. The call:

```
>> MatZ19(4, 4, [[1, 2], [2]])
```

```
          +-                                          -+
          |  1 mod 19, 2 mod 19, 0 mod 19, 0 mod 19   |
          |                                           |
          |  2 mod 19, 0 mod 19, 0 mod 19, 0 mod 19   |
          |                                           |
          |  0 mod 19, 0 mod 19, 0 mod 19, 0 mod 19   |
          |                                           |
          |  0 mod 19, 0 mod 19, 0 mod 19, 0 mod 19   |
          +-                                          -+
```

fixes the dimension of the matrix. Missing entries and inner lists are treated as zero components and zero rows, respectively.

An error message is issued if one of the given entries cannot be converted into an element over $\mathbb{Z}_{19}$:

```
>> MatZ19([[2, 3], [-1, I]])
```

```
 Error: unable to define matrix over Dom::IntegerMod(19) \
 [(Dom::Matrix(Dom::IntegerMod(19)))::new]
```

Example 5. This example illustrates how to create a matrix with components given as values of an index function. First we create the 2×2 Hilbert matrix (see also the functions `linalg::hilbert` and `linalg::invhilbert`):

```
>> matrix(2, 2, (i, j) -> 1/(i + j - 1))
```

$$\begin{bmatrix} 1 & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{3} \end{bmatrix}$$

Note the difference when working with expressions and functions. If you give an expression it is treated as a function in the row and column indices:

```
>> delete x:
   matrix(2, 2, x), matrix(2, 2, (i, j) -> x)
```

$$\begin{bmatrix} x(1, 1) & x(1, 2) \\ x(2, 1) & x(2, 2) \end{bmatrix}, \begin{bmatrix} x & x \\ x & x \end{bmatrix}$$

Example 6. Diagonal matrices can be created with the option `Diagonal` and a list of diagonal components:

```
>> MatC := Dom::Matrix(Dom::Complex):
   MatC(3, 4, [1, 2, 3], Diagonal)
```

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 3 & 0 \end{bmatrix}$$

Hence, to define the $n \times n$ identity matrix, you can enter:

```
>> MatC(3, 3, [1 $ 3], Diagonal)
```

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

or even call:

```
>> MatC(3, 3, x -> 1, Diagonal)
```

```
+-          -+
|  1, 0, 0   |
|            |
|  0, 1, 0   |
|            |
|  0, 0, 1   |
+-          -+
```

The easiest way to create the identity matrix, however, is to use the method `"identity"`:

```
>> MatC::identity(3)
```

```
+-          -+
|  1, 0, 0   |
|            |
|  0, 1, 0   |
|            |
|  0, 0, 1   |
+-          -+
```

Example 7. Toeplitz matrices can be defined with the option *`Banded`*. The following call defines a three-banded matrix with the component 2 on the main diagonal and the component -1 on the first subdiagonals:

```
>> matrix(4, 4, [-1, 2, -1], Banded)
```

```
+-                  -+
|   2, -1,  0,  0   |
|                   |
|  -1,  2, -1,  0   |
|                   |
|   0, -1,  2, -1   |
|                   |
|   0,  0, -1,  2   |
+-                  -+
```

Example 8. Some system functions can be applied to matrices, such as `norm`, `expand`, `diff`, `conjugate`, or `exp`.

For example, to expand the components of the matrix:

```
>> delete a, b:
   A := matrix(
     [[(a - b)^2, a^2 + b^2], [a^2 + b^2, (a - b)*(a + b)]]
   )
```

```
+-                                -+
|          2       2    2        |
|  (a - b) ,     a  + b           |
|                                  |
|    2    2                        |
|   a  + b , (a + b) (a - b)       |
+-                                -+
```

enter:

```
>> expand(A)
```

```
+-                                -+
|             2    2   2    2     |
|  - 2 a b + a  + b , a  + b       |
|                                  |
|        2    2        2    2     |
|       a  + b ,      a  - b       |
+-                                -+
```

If you want to differentiate the matrix components, then call for example:

```
>> diff(A, a)
```

```
+-                  -+
|  2 a - 2 b, 2 a    |
|                    |
|     2 a,    2 a    |
+-                  -+
```

To substitute matrix components by some values, enter:

```
>> subs(A, a = 1, b = -1)
```

```
+-      -+
|  4, 2  |
|        |
|  2, 0  |
+-      -+
```

The function `zip` can also be applied to matrices. The following call combines two matrices A and B by dividing each component of A by the corresponding component of B:

```
>> A := matrix([[4, 2], [9, 3]]):
   B := matrix([[2, 1], [3,-1]]):
   zip(A, B, '/')
```

```
+-       -+
|  2,  2  |
|         |
|  3, -3  |
+-       -+
```

The quoted character `'/'` is another notation for the function `_divide`, the functional form of the division operator `/`.

If one needs to apply a function to the components of a matrix, then use the function `map`. For example, to simplify the components of the matrix:

```
>> C := matrix(
     [[sin(x)^2 + cos(x)^2, exp(x) - exp(x/2)^2],
     [(a^2 - b^2)/(a + b), 1]]
   )
```

```
+-                                        -+
|        2         2              / x \2  |
|  cos(x)  + sin(x) , exp(x) - exp| - |   |
|                                 \ 2 /   |
|                                         |
|        2    2                           |
|       a  - b                            |
|       -------,              1           |
|        a + b                            |
+-                                        -+
```

call:

```
>> map(C, simplify)
```

```
+-          -+
|    1,   0  |
|            |
|  a - b, 1  |
+-          -+
```

Example 9. A column vector is represented as a 2×1 matrix:

```
>> MatR := Dom::Matrix(Dom::Real):
   v := MatR(2, 1, [1, 2])
```

```
+-   -+
|  1  |
|     |
|  2  |
+-   -+
```

The dimension of this vector is:

```
>> MatR::matdim(v)
```

```
                                [2, 1]
```

Use `linalg::vecdim`, or even call `nops(v)` to get the length of a vector:

```
>> linalg::vecdim(v)
```

```
                                   2
```

The *i*th component of this vector can be extracted in two ways: either by `v[i,1]` or by `v[i]`:

```
>> v[1], v[2]
```

```
                                1, 2
```

We get the 2-norm of `v` by the following call:

```
>> norm(v, 2)
```

```
                                 1/2
                                5
```

Super-Domain: `Dom::BaseDomain`

Axioms

if `R` has `Ax::canonicalRep`
`Ax::canonicalRep`

Changes:

- The method `"dimen"` was renamed to `"matdim"`.
- The method `"newThis"` was renamed to `"create"`.
- `"_invert"` now uses the function `numeric::inverse` for certain component rings `R`.
- New method `"diff"` which applies the function `diff` to the components of a matrix.
- New method `"evalp"` for matrices over polynomial rings.
- `exp` now uses the function `numeric::expMatrix` for a floating-point approximation of the exponential of a matrix, i.e., if the component ring `R` is the domain `Dom::Float`.

- New method `"expand"` which applies the function `expand` to the components of a matrix.
- New method `"factor"` for rewriting the matrix A in the form $A = sB$ with a scalar s.
- New method `"float"` for computing a floating-point approximation of matrix components.
- New method `"identity"` to ease the construction of identity matrices.
- New method `"normal"` for simplification of matrix components.
- New method `"unapply"` for overloading the function `fp::unapply`.

`Dom::MatrixGroup` – the Abelian group of $m \times n$ matrices

`Dom::MatrixGroup(m, n, R)` creates the Abelian group of $m \times n$ matrices over the component ring R.

Domain:

- `Dom::MatrixGroup(m, n<, R>)`

Parameters:

`m, n` — positive integers (matrix dimension)
`R` — a commutative ring, i.e., a domain of category `Cat::CommutativeRing`; default is `Dom::ExpressionField()`

Details:

- `Dom::MatrixGroup(m, n, R)` creates a domain which represents the Abelian group of $m \times n$ matrices over the component ring `R`, i.e., it is a domain of category `Cat::AbelianGroup`.
- The domain `Dom::ExpressionField()` is used as the component ring for the matrices if the optional parameter `R` is not given.
- For matrices of a domain created by `Dom::MatrixGroup(m, n, R)`, matrix arithmetic is implemented by overloading the standard arithmetical operators `+, -, *, /` and `^`. All functions of the `linalg` package dealing with matrices can be applied.
- `Dom::MatrixGroup(m, n, R)` has the domain `Dom::Matrix(R)` as its super domain, i.e., it inherits each method which is defined by `Dom::Matrix(R)` and not re-implemented by `Dom::MatrixGroup(m, n, R)`.

  Methods described below are implemented by `Dom::MatrixGroup`.

- The domain `Dom::Matrix(R)` represents matrices over `R` of arbitrary size, and it therefore does not have any algebraic structure (except of being a *set* of matrices).

  The domain `Dom::SquareMatrix(n, R)` represents the *ring* of $n \times n$ matrices over R.

Creating Elements:

- `Dom::MatrixGroup(m, n, R)(Array)`
- `Dom::MatrixGroup(m, n, R)(Matrix)`
- `Dom::MatrixGroup(m, n, R)(<m, n>)`
- `Dom::MatrixGroup(m, n, R)(<m, n, >List)`
- `Dom::MatrixGroup(m, n, R)(<m, n, >ListOfRows)`
- `Dom::MatrixGroup(m, n, R)(<m, n, >f)`
- `Dom::MatrixGroup(m, n, R)(<m, n, >List, Diagonal)`
- `Dom::MatrixGroup(m, n, R)(<m, n, >g, Diagonal)`
- `Dom::MatrixGroup(m, n, R)(<m, n, >List, Banded)`

Parameters:

`Array`	—	an `m`×`n` array
`Matrix`	—	an `m`×`n` matrix, i.e., an element of a domain of category `Cat::Matrix`
`List`	—	a list of matrix components
`ListOfRows`	—	a list of at most `m` rows; each row is a list of at most `n` matrix components
`f`	—	a function or a functional expression with two parameters (the row and column index)
`g`	—	a function or a functional expression with one parameter (the row index)

Options:

`Diagonal`	—	create a diagonal matrix
`Banded`	—	create a banded Toeplitz matrix

Categories:

`Cat::Matrix(R)`, `Cat::AbelianGroup`

if `R` has `Cat::Field`, then
`Cat::VectorSpace(R)`

Related Domains: `Dom::Matrix`, `Dom::SquareMatrix`

Details:

- `Dom::MatrixGroup(m, n, R)(Array)` and `Dom::MatrixGroup(m, n, R)(Matrix)` create a new matrix formed by the entries of `Array` and `Matrix`, respectively.

  The components of `Array` and `Matrix`, respectively, are converted into elements of the domain `R`. An error message is issued if one of these conversions fails.

- The call `Dom::MatrixGroup(m, n, R)(<m, n>)` returns the $m \times n$ zero matrix. Note that the $m \times n$ zero matrix can also be found in the entry `"zero"` (see below).

- `Dom::MatrixGroup(m, n, R)(<m, n, >List)` creates an $m \times n$ matrix with components taken from the list `List`.

  This call is only allowed for $m \times 1$ or $1 \times n$ matrices, i.e., if either `m` or `n` is equal to one.

  If the list has too few entries, the remaining components of the matrix are set to zero.

  The entries of the list are converted into elements of the domain `R`. An error message is issued if one of these conversions fails.

- `Dom::MatrixGroup(m, n, R)(<m, n, >ListOfRows)` creates an $m \times n$ matrix with components taken from the nested list `ListOfRows`. Each inner list corresponds to a row of the matrix.

  If an inner list has less than `n` entries, the remaining components in the corresponding row of the matrix are set to zero. If there are less than `m` inner lists, the remaining lower rows of the matrix are filled with zeroes.

  The entries of the inner lists are coerced into elements of the domain `R`. An error message is issued if one of these conversions fails.

- `Dom::MatrixGroup(m, n, R)(<m, n, >f)` returns the matrix whose (i, j)th component is the value of the function call `f(i, j)`. The row index i ranges from 1 to `m` and the column index j from 1 to `n`.

  The function values are coerced into elements of the domain `R`. An error message is issued if one of these conversions fails.

Option <*Diagonal*>:

- With the option *Diagonal*, diagonal matrices can be created with diagonal elements taken from a list, or computed by a function.

- `Dom::MatrixGroup(m, n, R)(<m, n, >List, Diagonal)` creates the $m \times n$ diagonal matrix whose diagonal elements are the entries of `List`.

`List` must have at most $\min(m, n)$ entries. If it has fewer elements, then the remaining diagonal elements are set to zero.

The entries of `List` are coerced into elements of the domain `R`. An error message is issued if one of these conversions fails.

- `Dom::MatrixGroup(m, n, R)(<m, n, >g, Diagonal)` returns the matrix whose ith diagonal element is `g(i)`, where the index i runs from 1 to $\min(m, n)$.

  The function values are coerced into elements of the domain `R`. An error message is issued if one of these conversions fails.

Option <*Banded*>:

- With the option *Banded*, banded matrices can be created.

  A *banded matrix* has all entries zero outside the main diagonal and some of the adjacent sub- and superdiagonals.

- `Dom::MatrixGroup(m, n, R)(<m, n, >List, Banded)` creates an $m \times n$ banded Toeplitz matrix with the elements of `List` as entries. The number of entries of `List` must be odd, say $2h + 1$, and must not exceed `n`. The resulting matrix has bandwidth at most $2h + 1$.

  All elements of the main diagonal of the created matrix are initialized with the middle element of `List`. All elements of the ith subdiagonal are initialized with the $(h + 1 - i)$th element of `List`. All elements of the ith superdiagonal are initialized with the $(h + 1 + i)$th element of `List`. All entries on the remaining sub- and superdiagonals are set to zero.

  The entries of `List` are converted into elements of the domain `R`. An error message is issued if one of these conversions fails.

Entries:

`one`	is only defined if `m` is equal to `n`; in that case it defines the $n \times n$ identity matrix.
`randomDimen`	is set to `[m, n]`.
`zero`	is the $m \times n$ zero matrix.

Mathematical Methods

Method `evalp`: evaluating matrices of polynomials at a certain point

```
evalp(dom A, equation x = a, ...)
```

- This method evaluates the polynomial components of `A` at the point `x = a`. See the system function `evalp` for details.

  The matrix returned is of the domain `Dom::MatrixGroup(m, n, R::coeffRing)`, if the evaluation of each component leads to an element of the coefficient ring of the polynomial domain. Otherwise the matrix returned is of the domain of `A`.
- This method is only defined if `R` is a polynomial ring of category `Cat::Polynomial`.
- This method overloads the function `evalp` for matrices, i.e., one may use it in the form `evalp(A, x = a)`.

Method `identity`: identity matrix

`identity(`*`positive integer`* `k)`

- This method returns the $k \times k$ identity matrix.
- The matrix returned is of the domain `Dom::Matrix(R)`, if `m`$\neq$`n` or if `k`$\neq$`n`.

Method `matdim`: matrix dimension

`matdim(`*`dom`* `A)`

- This method returns the list `[m, n]`, i.e., the matrix dimension of `A`.

Method `random`: random matrix generation

`random()`

- This method returns a random $m \times n$ matrix.
- The components of the random matrix are randomly generated with the method `"random"` of the component ring `R`.

Access Methods

Method `_concat`: horizontally concatenation of matrices

`_concat(`*`dom`* `A, `*`dom`* `B, ...)`

- This method appends the matrices `B, ...` to the right side of the matrix `A`.
- An error message is issued if the given matrices do not have the same number of rows.
- The returned matrix is of the domain `Dom::Matrix(R)`.

- This method overloads the function `_concat` for matrices, i.e., one may use it in the form `A . B . ...`, or in functional notation: `_concat(A, B, ...)`.

Method `_index`: matrix indexing

`_index(`*`dom`* `A, `*`row index`* `i, `*`column index`* `j)`

- Returns the (i, j)-th entry of the matrix `A`.
- This method overloads the function `_index` for matrices, i.e., one may use it in the form `A[i, j]` or in functional notation: `_index(A, i, j)`.

`_index(`*`dom`* `A, `*`row-range`* `r1..r2, `*`column-range`* `c1..c2)`

- Returns the submatrix of `A`, created by the rows of `A` with indices from `r1` to `r2` and the columns of `A` with indices from `c1` to `c2`.
- The submatrix is of the domain `Dom::Matrix(R)`.

`_index(`*`dom`* `A, `*`index`* `i)`

- This method returns the ith entry of `A`.
- This call is only allowed for $1 \times n$ or $m \times 1$ matrices, i.e., either `m` or `n` must be equal to one. Otherwise an error message is issued.

`_index(`*`dom`* `A, `*`index-range`* `i1..i2)`

- This method returns the subvector of `A`, formed by the entries with index `i1` to `i2` (see also the method `"op"`).
- This call is only allowed for $1 \times n$ or $m \times 1$ matrices, i.e., either `m` or `n` must be equal to one. Otherwise an error message is issued.
- This method overloads the function `_index` for matrices, i.e., one may use it in the form `A[i,j]`, `A[r1..r2,c1..c2]`, `A[i]` or `A[i1..i2]`, respectively, or in functional notation: `_index(A, ...)`.

Method `concatMatrix`: horizontally concatenation of matrices

`concatMatrix(`*`dom`* `A, `*`dom`* `B, ...)`

- This method is identical to the method `"_concat"`.

Method `col`: extracting a column

`col(`*`dom`* `A, `*`column index`* `c)`

- This method extracts the column with index `c` of the matrix `A` and returns it as a column vector, i.e., as an element of the domain `Dom::Matrix(R)`.
- An error message is issued if `c` is less than one or greater than `n`.

Method `delCol`: deleting a column

`delCol(`*`dom`* `A, `*`column index`* `c)`

- This method returns the matrix obtained by deleting the column with index `c` of the matrix `A`.
- `NIL` is returned if `A` only consists of one column.
- The returned matrix is of the domain `Dom::Matrix(R)`.

- An error message is issued if `c` is less than one or greater than `n`.

Method `delRow`: deleting a row

`delRow(`*`dom`* `A, `*`row index`* `r)`

- This method returns the matrix obtained by deleting the row with index `r` of the matrix `A`.
- `NIL` is returned if `A` only consists of one row.
- The returned matrix is of the domain `Dom::Matrix(R)`.

- An error message is issued if `r` is less than one or greater than `m`.

Method `row`: extracting a row

`row(`*`dom`* `A, `*`row index`* `r)`

- This method extracts the row with index `r` of the matrix `A` and returns it as a row vector, i.e., as an element of domain `Dom::Matrix(R)`.
- An error message is issued if `r` is less than one or greater than `m`.

Method `stackMatrix`: concatenating of matrices vertically

`stackMatrix(`*`dom`* `A, `*`dom`* `B, ...)`

- This method stacks the matrix `A` on the top of the matrix `B`. If further arguments are given, then the result is stacked on the top of the third matrix, and so on.
- An error message is issued if the given matrices do not have the same number of columns.
- The matrix returned is of the domain `Dom::Matrix(R)`.

Conversion Methods

Method `convert`: conversion into a matrix

`convert(`*`any`* `x)`

- This method tries to convert `x` into a matrix of type `Dom::MatrixGroup(m, n, R)`.
- `FAIL` is returned if the conversion fails.
- `x` may either be an `m`×`n` array, or an `m`×`n` matrix of category `Cat::Matrix`. `x` can also be a list. See the parameter `List` and `ListOfRows` in "Creating Elements" above for admissible values of `x`.
  The entries of `x` must be convertable into elements of the domain `R`, otherwise `FAIL` is returned.

Example 1. A lot of examples can be found on the help page of the domain constructor `Dom::Matrix`, and most of them are also examples for working with domains created by `Dom::MatrixGroup`. This example only highlights some differences with respect to working with matrices of the domain `Dom::Matrix(R)`.

The following command defines the abelian group of 3×4 matrices over the rationals:

```
>> MatGQ := Dom::MatrixGroup(3, 4, Dom::Rational)

              Dom::MatrixGroup(3, 4, Dom::Rational)

>> MatGQ::hasProp(Cat::AbelianGroup), MatGQ::hasProp(Cat::Ring)

                              TRUE, FALSE
```

`MatGQ` is a commutative group with respect to the addition of matrices. The unit of this group is the 3×4 zero matrix:

```
>> MatGQ::zero

                          +-             -+
                          |  0, 0, 0, 0   |
                          |               |
                          |  0, 0, 0, 0   |
                          |               |
                          |  0, 0, 0, 0   |
                          +-             -+
```

Note that some operations defined by the domain `MatGQ` return matrices which are no longer elements of the matrix group. They return matrices of the domain `Dom::Matrix(Dom::Rational)`, the super-domain of `MatGQ`.

For example, if we define the matrix:

```
>> A := MatGQ([[1, 2, 1, 2], [-5, 3], [2, 1/3, 0, 1]])
```

```
                          +-                 -+
                          |   1,  2,  1, 2   |
                          |                  |
                          |  -5,  3,  0, 0   |
                          |                  |
                          |   2, 1/3, 0, 1   |
                          +-                 -+
```

and delete its third column, we get the matrix:

```
>> MatGQ::delCol(A, 3)
```

```
                           +-             -+
                           |   1,  2,  2   |
                           |               |
                           |  -5,  3,  0   |
                           |               |
                           |   2, 1/3, 1   |
                           +-             -+
```

which is of the domain type:

```
>> domtype(%)
```

```
                      Dom::Matrix(Dom::Rational)
```

As another example we create the 3×3 identity matrix using the method `"identity"` of our domain:

```
>> E3 := MatGQ::identity(3)
```

```
                             +-          -+
                             |  1, 0, 0  |
                             |           |
                             |  0, 1, 0  |
                             |           |
                             |  0, 0, 1  |
                             +-          -+
```

This is also a matrix of the domain `Dom::Matrix(Dom::Rational)`:

```
>> domtype(E3)
```

```
                      Dom::Matrix(Dom::Rational)
```

If we concatenate `E3` to the right of the matrix `A` defined above, we get the 3×7 matrix:

```
>> B := A . E3
```

```
+-                              -+
|   1,  2,  1, 2, 1, 0, 0  |
|                              |
|  -5,  3,  0, 0, 0, 1, 0  |
|                              |
|   2, 1/3, 0, 1, 0, 0, 1  |
+-                              -+
```

which is of the domain type `Dom::Matrix(Dom::Rational)`:

```
>> domtype(B)
```

```
                    Dom::Matrix(Dom::Rational)
```

Example 2. We can convert a matrix from a domain created with `Dom::MatrixGroup` into or from another matrix domain, as shown next:

```
>> MatGR := Dom::MatrixGroup(2, 3, Dom::Real):
   MatC := Dom::Matrix(Dom::Complex):

>> A := MatGR((i, j) -> i*j)
```

```
                            +-         -+
                            |  1, 2, 3  |
                            |           |
                            |  2, 4, 6  |
                            +-         -+
```

To convert `A` into a matrix of the domain `MatC`, enter:

```
>> coerce(A, MatC)
```

```
                            +-         -+
                            |  1, 2, 3  |
                            |           |
                            |  2, 4, 6  |
                            +-         -+
```

```
>> domtype(%)
```

```
                      Dom::Matrix(Dom::Complex)
```

The conversion is done component-wise. For example, we define the following matrix:

```
>> B := MatC([[0, 1, 0], [exp(I), 0, 1]])
```

```
+-               -+
|     0,   1, 0  |
|                |
|  exp(I), 0, 1  |
+-               -+
```

The matrix B has one complex component and therefore cannot be converted into the domain `MatGR`:

```
>> coerce(B, MatGR)

                                FAIL
```

Note: The system function `coerce` uses the methods `"convert"` and `"convert_to"` implemented by any domain created with `Dom::MatrixGroup` and `Dom::Matrix`.

Super-Domain: `Dom::Matrix`

Axioms

if `R` has `Ax::canonicalRep`, then
`Ax::canonicalRep`

Changes:

- The method `"dimen"` was renamed to `"matdim"`.
- The method `"newThis"` was renamed to `"create"`.
- Some new methods were implemented or extended for the domain `Dom::Matrix`. See the corresponding help page for details. Note that `Dom::MatrixGroup(m, n, R)` inherits every method which is defined for `Dom::Matrix(R)` and not re-implemented by `Dom::MatrixGroup(m, n, R)`.

`Dom::MonomOrdering` – monomial orderings

`Dom::MonomOrdering` represents the set of all possible monomial orderings. A monomial ordering is a well-ordering of the set of all k-tuples of nonnegative integers for some k.

Domain:

- `Dom::MonomOrdering`

Details:

- In MuPAD, a monomial ordering is implemented as a function that, when applied to two lists of nonnegative integers, returns `-1`, `0`, or `1` if the first list is respectively smaller than, equal to, or greater than the second list. Each ordering can only compare lists of one fixed length, called its its *order length*. Since the lists under consideration will be exponent vectors in most cases, their length is also referred to as the number of indeterminates.
- Monomial orderings are used in algebraic geometry for comparing terms $\prod_{i=1}^{n} X_i^{\alpha_i}$ and $\prod_{i=1}^{n} X_i^{\beta_i}$ in a polynomial ring. Since `Dom::MonomOrdering` works on the exponent vectors $[\alpha_1, \ldots, \alpha_n]$ and $[\beta_1, \ldots, \beta_n]$, `degreevec` must be applied to the terms to be compared before applying `Dom::MonomOrdering`.
- Elements of `Dom::MonomOrdering` can be used as arguments for `lcoeff`, `lmonomial`, `lterm`, and `tcoeff` as well as for the functions of the `groebner` package in order to specify the monomial ordering to be considered.

Monomial orderings are created by calling `Dom::MonomOrdering(someIdentifier(parameters`
where `someIdentifier` is one of a certain set of predefined identifiers, as stated below. Converting `someIdentifier` into a string gives the *order type* of the monomial ordering.

Creating Elements:

- `Dom::MonomOrdering(Lex(n))`
- `Dom::MonomOrdering(RevLex(n))`
- `Dom::MonomOrdering(DegLex(n))`
- `Dom::MonomOrdering(DegRevLex(n))`
- `Dom::MonomOrdering(DegInvLex(n))`
- `Dom::MonomOrdering(WeightedLex(w1, ...,wn))`
- `Dom::MonomOrdering(WeightedRevLex(w1, ...,wn))`
- `Dom::MonomOrdering(WeightedDegLex(w1, ..,wn))`
- `Dom::MonomOrdering(WeightedDegRevLex(w1, ..,wn))`
- `Dom::MonomOrdering(Block(o1, ...))`
- `Dom::MonomOrdering(Matrix(params))`

Parameters:

`n`	—	positive integer
`w1, ...`	—	numerical expressions
`o1,..`	—	valid arguments to `Dom::MonomOrdering`
`params`	—	a sequence valid as the sequence of arguments to `Dom::Matrix()`.

Categories:

`Cat::BaseCategory`

Details:

- `Dom::MonomOrdering(Lex(n))` creates the lexicographical order on n indeterminates.
- `Dom::MonomOrdering(RevLex(n))` creates the reverse lexicographical order on n indeterminates, i.e., `Dom::MonomOrdering(RevLex(n))([a1,...,an]) = Dom::MonomOrdering(Lex(n))([an,...,a1])`.
- `Dom::MonomOrdering(DegLex(n))` creates the degree order on n indeterminates with the lexicographical order used for tie-break.
- `Dom::MonomOrdering(DegRevLex(n))` creates the degree order on n indeterminates with the reverse lexicographical order used for tie-break .
- `Dom::MonomOrdering(DegInvLex(n))` creates the degree order on n indeterminates, with the tie break being the opposite to the lexicographical order.
- `Dom::MonomOrdering(Weighted...(w1,...,wn))` returns a weighted degree order with weights w_1 through w_n. The word following the word `Weighted` specifies the tie-break used. Note that MuPAD uses the ordinary degree order as the first tie-break.
- `Dom::MonomOrdering(Matrix(params))` creates a matrix order, with the order matrix defined by `Dom::Matrix()(params)`.
- `Dom::MonomOrdering(Block(o1, ..., on))` or, equivalently, `Dom::MonomOrdering([o1, ..., on])`, creates a block order such that `Dom::MonomOrdering(o1)` is used on the first indeterminates, then `Dom::MonomOrdering(o2)` is used as a tie-break on the following indeterminates etc.

  Block orders may be nested, i.e., the blocks may be block orders, too.
- Weight vectors with negative entries and order matrices do not define well-orderings in general. You may enter such orderings, but it may cause trouble, e.g., to use them with the `groebner` package.

Mathematical Methods

Method `func_call`: compare two lists of integers

```
func_call(dom o, list l1, list l2)
```

- This method is called by entering `o(l1, l2)`. It returns `-1` if $l1 < l2$, `1` if $l1 > l2$, and `0` if $l1 = l2$.
- The lengths of `l1` and `l2` must not exceed the order length of `o`. If `l1` or `l2` is too short, the necessary number of zeroes is appended.

Access Methods

Method `ordertype`: return the type of an order

```
ordertype(dom o)
```

- This method returns the order type of `o`.
- If `o` equals `Dom::MonomOrdering(someIdentifier(params))`, then converting `someIdentifier` into a string gives the order type of `o`.

Method `orderlength`: return the length of an order

```
orderlength(dom o)
```

- This method returns length of `o`; this is the largest integer k for which `o` works on lists of length k.

Method `nops`: number of blocks

```
nops(dom o)
```

- A block order `Dom::MonomOrdering(Block(o1,...,on))` is said to have n blocks. An order of any other type is said to have one block.

Method `block`: get a particular block

```
block(dom o, positive integer i)
```

- This method returns the i-th block of `o`, or `FAIL` if the order `o` does not have that many blocks.

Method `blocktype`: get the order type of a particular block

`blocktype(`*`dom`* `o, `*`positive integer`* `i)`

- This method returns the order type of the i-th block of `o`.

Method `blocklength`: get the order length of a particular block

`blocklength(`*`dom`* `o, `*`positive integer`* `i)`

- This method returns the order length of the i-th block of `o`.

Conversion Methods

Method `expr`: return an expression from which the order can be restored

`expr(`*`dom`* `o)`

- This method returns an expression `someIdentifier(parameters)` such that applying `Dom::MonomOrdering` to it would give back `o`.

Example 1. We define `ORD` by prescribing that lists $[a, b, c]$ are ordered according to their weighted degrees $5a + 2b + \pi c$. For lists with equal weighted degree, the non-weighted degree $a + b + c$ is used as a tie-break. Finally, the lexicographical order decides (in fact, this last step is not necessary because π is irrational).

```
>> ORD:=Dom::MonomOrdering(WeightedDegLex(5, 2, PI))
                    WeightedDegLex(5, 2, PI)
```

With respect to `ORD`, `[1, 6, 1]` is smaller than `[2, 1, 3]`:

```
>> ORD([1,6,1], [2,1,3])
                                -1
```

Super-Domain: `Dom::BaseDomain`

Changes:

- `Dom::MonomOrdering` is a new domain.

`Dom::Multiset` – multisets

`Dom::Multiset` is the domain of multisets, i.e., sets with possibly multiple identical elements.

Details:

- A multiset is represented by a set of lists of the form $[s, m]$, where s is an element of the multiset and m its multiplicity.
- Multisets can be returned by the system solver `solve`. For example, the input `solve(x^3 - 4*x^2 + 5*x - 2, x, Multiple)` gives all roots of the polynomial $x^3 - 4x^2 + 5x - 2$ in form of the multiset `{[1, 2], [2, 1]}`.
- The standard set operations such as union, intersection and subtraction of sets have been extended to deal with multisets.

  These operations can handle different types of sets, such as sets of type `DOM_SET` and multisets. One may, for example, compute the union of the multiset `{[a, 2], [b, 1]}` and the set `{c}`, which results in the multiset `{[a, 2], [b, 1], [c, 1]}`.
- The elements of the multiset are sorted at the time where the multiset is created. The system function `sort` is used in order to guarantee that exactly one representation exists for a multiset, independent of the sequence in which the arguments appear.

Creating Elements:

- `Dom::Multiset(<s1, s2, ...>)`

Parameters:

`s1, s2, ...` — objects of any type

Categories:

`Cat::Set`

Related Domains: `DOM_SET`, `Dom::ImageSet`

Details:

- `Dom::Multiset(s1, s2, ...)` creates the multiset consisting of the elements `s1, s2, ...`
- Multiple identical elements in `s1, s2, ...` are collected. For example, the call `Dom::Multiset(a, b, a, c)` creates a multiset with the elements `a, b, c`. The element `a` has multiplicity two, the other two elements `b` and `c` both have multiplicity one.

Entries:

`isFinite` is `TRUE` because `Dom::Multiset` represents finite sets.

`inhomog_intersect` a table of the form `T = Proc(multiset, setoftypeT)`. This entry is used internally by the implementation, and thus should not be touched.

`inhomog_union` a table of the form `T = Proc(multiset, setoftypeT)`. This entry is used internally by the implementation, and thus should not be touched.

Mathematical Methods

Method `normal`: normalization of multisets

`normal(`*dom* `set)`

- This method normalizes every element of `set` using the system function `normal`.
- This method overloads the function `normal` for multisets, i.e., one may use it in the form `normal(set)`.

Method `powerset`: the power set of a multiset

`powerset(`*dom* `set)`

- This method computes the power set of the multiset `set`, i.e., all sub-multisets of `set`.
- The power set of `set` is returned as a set of multisets.

Method `random`: random multiset generation

`random()`

- This method returns a randomly generated multiset. It uses the function `random` to create the elements of the random multiset. Therefore, the elements of the returned multiset will be integers.
- The number of elements created, including their multiplicities, is restricted to 20.

Access Methods

Method `_index`: multiset indexing

```
_index(dom set, index i)
```

- Returns the i-th element s of the multiset `set` and its multiplicity m in form of the list $[s, m]$.
  Note that the elements of the multiset are sorted with the use of the system function `sort`, and thus the order of a multiset depends on the sorting criteria specified by this function.
- See the method `"op"`.
- This method overloads the function `_index` for multisets, i.e., one may use it in the form `set[i]`, or in functional notation: `_index(set, i)`.

Method `contains`: check on existence of set elements

```
contains(dom set, any s)
```

- This method returns `TRUE` if `set` does contain `s` (i.e., if `s` is an element of `set`), otherwise `FALSE` is returned. For the comparison of two elements, the system function `_equal` is used, which only tests for syntactical equivalence.
- This method overloads the function `contains` for multisets, i.e., one may use it in the form `contains(set, s)`.

Method `equal`: test on equality of multisets

```
equal(dom set1, dom set2)
```

- This method tests if the two multisets `set1` and `set2` are equal and returns `TRUE`, `FALSE` or `UNKNOWN`, respectively.
- The system function `_equal` is used for the test.

Method `expand`: expand a multiset to a sequence of its elements

```
expand(dom set)
```

- This method returns an expression sequence (i.e., an expression of type `"_exprseq"`) of all elements in `set`, appearing in correspondence to their multiplicity. For example, for the multiset $\{[1,1],[2,2],[3,1]\}$. the expression sequence $3,2,2,1$ is returned.
- This method overloads the function `expand` for multisets, i.e., one may use it in the form `expand(set)`.

Method `getElement`: extracts one element from a multiset

```
getElement(dom set)
```

- This method returns the first element of `set`.
- Note that the elements of the multiset are sorted with the use of the system function `sort`, and thus the order of a multiset depends on the sorting criteria specified by this function.
- This method overloads the function `solvelib::getElement`, i.e., one may use it in the form `solvelib::getElement(set)`.

Method `has`: check on existence of (sub-)expressions

```
has(dom set, any expr)
```

- This method returns `TRUE` if the multiset `set` has a subexpression equal to `expr`, and `FALSE` otherwise.
- To check whether `expr` is contained as an element of `set` and not as a subexpression of the elements of `set`, the function `contains` must be used.
- This method overloads the function `has` for multisets, i.e., one may use it in the form `has(set, expr)`.

Method `map`: apply a function to multiset elements

```
map(dom set, function func<, any expr, ...>)
```

- This method maps the function `func` onto the elements (not onto their multiplicities) of the multiset `set`, with the additional function parameters `expr, ...` passed to `func`, if given.

  See the system function `map` for details.
- It overloads the function `map` for multisets, i.e., one may use it in the form `map(set, func, ...)`.

Method `multiplicity`: multiplicity of an element

```
multiplicity(dom set, any s)
```

- This method returns the multiplicity of the element `s` in the multiset `set`.
- Elements which are not contained in `set` have multiplicity zero.

Method `nops`: number of different elements in a multiset

```
nops(dom set)
```

- This method returns the number of different elements in `set`.
- This method overloads the function `nops` for multisets, i.e., one may use it in the form `nops(set)`.

Method `op`: elements of a multiset

```
op(dom set, positive integer i)
```

- Returns the i-th element s of the multiset `set` and its multiplicity m in form of the list $[s, m]$.
- See also the method `"_index"`.
- Note that the elements of the multiset are sorted with the use of the system function `sort`, and thus the order of a multiset depends on the sorting criteria specified by this function.
- This method overloads the function `op` for multisets, i.e., one may use it in the form `op(s, i)`.

Method `select`: selecting of multiset elements

```
select(dom set, function func<, any expr, ...>)
```

- This method maps the function `func` onto the elements (not onto their multiplicities) of the multiset `set`, with the additional function parameters `expr, ...` passed on to `func`, if given, and returns a multiset with those elements for which the function call returned `TRUE`.
- This method overloads the function `select` for multisets, i.e., one may use it in the form `select(set, func, ...)`. See `select` for details.

Method `split`: splitting a multiset

```
split(dom set, function func<, any expr, ...>)
```

- This method maps the function `func` onto the elements (not onto their multiplicities) of the multiset `set`, with the additional function parameters, if given, `expr, ...` passed to `func` and returns a list of three multisets with those elements for which the function call returned `TRUE`, `FALSE`, and `UNKNOWN` respectively.
- This method overloads the function `split` for multisets, i.e., one may use it in the form `split(set, func, ...)`. See `split` for details.

Method `subs`: substitution of elements in multisets

```
subs(dom set, ...)
```

- This method applies the function `subs` with additionally given parameters to the elements (not onto their multiplicities) of the multiset `set`.

  See the system function `subs` for details.
- This method overloads the function `subs` for multisets, i.e., one may use it in the form `subs(set, ...)`.

Conversion Methods

Method `convert`: conversion into a multiset

```
convert(any x)
```

- This method tries to convert `x` into a multiset of domain type `Dom::Multiset`.
- `FAIL` is returned if the conversion fails.
- Currently only sets of type `DOM_SET` can be converted into multisets.

Method `convert_to`: multiset conversion

```
convert_to(dom set, any T)
```

- This method tries to convert the multiset `set` into an element of domain type `T`.
- `FAIL` is returned if the conversion fails.
- Currently `T` may either be `DOM_SET` to convert the multiset `set` into a set (loosing the multiplicities and the order of the elements of `set`), or `DOM_EXPR` or `"_exprseq"` to convert `set` into an expression sequence (see the method `"expand"` for details).
- See also the method `"expr"`.

Method `expr`: multiset conversion into an object of a kernel domain

```
expr(dom set)
```

- This method converts `set` into a set of type `DOM_SET` consisting of lists of the form $[s, m]$, where s is an element of `set` and m its multiplicity.
- This method overloads the function `expr` for multisets, i.e., one may use it in the form `expr(set)`.

Technical Methods

Method `bin_intersect`: intersection of two multisets

`bin_intersect(`*`dom`* `set1, `*`dom`* `set2)`

- Computes the intersection of `set1` and `set2`.
- This method is called from routines defined in the category `Cat::Set`, which implements among others the overloading of the function `_intersect` for multisets. One may intersect two multisets directly by `set1 intersect set2`, or in functional notation by `_intersect(set1, set2)`.

Method `bin_minus`: subtraction of two multisets

`bin_minus(`*`dom`* `set1, `*`dom`* `set2)`

- Computes `set1` minus `set2`.
- This method is called from routines defined in the category `Cat::Set`, which implements among others the overloading of the function `_minus` for multisets. One may subtract two multisets directly by `set1 minus set2`, or in functional notation by `_minus(set1, set2)`.

Method `homog_union`: union of multisets

`homog_union(`*`dom`* `set, ...)`

- Computes the union of the given multisets.
- This method is called from routines defined in the category `Cat::Set`, which implements among others the overloading of the function `_union` for multisets. One may compute the union of two multisets directly by `set1 union set2`, or in functional notation by `_union(set1, set2)`.

Method `nested_union`: union of nested sets

`nested_union(`*`set`* `setofsets)`

- This method computes the union of the sets in `setofsets`. The contained sets may be multisets or sets of type `DOM_SET`.
- This method is called from routines defined in the category `Cat::Set`, which implements among others the overloading of the function `_union` for multisets and sets. One may compute the union of multisets and sets directly by `set1 union set2`, or in functional notation by `_union(set1, set2)`.

Example 1. The multiset $\{a, a, b\}$ consists of the two different elements a and b, where a has multiplicity two and b has multiplicity one:

```
>> delete a, b, c:
   set1 := Dom::Multiset(a, a, b)

                          {[a, 2], [b, 1]}
```

We create another multiset:

```
>> set2 := Dom::Multiset(a, c, c)

                          {[a, 1], [c, 2]}
```

Standard set operations such as union, intersection or subtraction are implemented for multisets and can be performed using the standard set operators of MuPAD:

```
>> set1 union set2

                     {[b, 1], [a, 3], [c, 2]}

>> set1 intersect set2

                              {[a, 1]}

>> contains(set1, a), contains(set1, d)

                             TRUE, FALSE
```

Example 2. Some system functions were overloaded for multisets, such as `expand`, `normal` or `split`.

If we apply `expand` to a multiset, for example, we get an expression sequence of all elements of the multiset (appearing in correspondence to their multiplicity):

```
>> delete a, b, c, d, e:
   set := Dom::Multiset(a, b, c, a, c, d, c, e, c)

              {[a, 2], [b, 1], [d, 1], [e, 1], [c, 4]}

>> expand(set)

                     e, d, c, c, c, c, b, a, a
```

If you want to convert a multiset into an ordinary set of the domain type `DOM_SET`, use `coerce`:

```
>> coerce(set, DOM_SET)
```

```
{a, b, c, d, e}
```

Note: The system function `coerce` uses the methods `"convert"` and `"convert_to"` of the domain `Dom::Multiset`.

Compare the last result with the return value of the function `expr`, when it is applied for multisets:

```
>> expr(set)
```

```
{[a, 2], [b, 1], [d, 1], [e, 1], [c, 4]}
```

The result is a set of the domain type `DOM_SET`, consisting of lists of the domain type `DOM_LIST` with two entries, an element of the multiset and the corresponding multiplicity of that element.

Super-Domain: `Dom::BaseDomain`

Changes:

- Method `contains` now returns either `TRUE` or `FALSE` instead of an integer.
- Method `convert` expects exactly one argument only and was extended to deal with sets of type `DOM_SET`.
- New entries and methods: `"isFinite"`, `"convert_to"`, `"equal"`, `"getElement"`, `"powerset"`, `"random"`.
- The methods `"_intersect"`, `"_subtract"` and `"_union"` were removed and replaced by the following new entries and methods: `"bin_intersect"`, `"bin_minus"`, `"inhomog_intersect"`, `"inhomog_union"`, `"homog_union"` and `"nested_union"`.
- The methods `"has"`, `"map"`, `"select"`, and `"subs"` now work on the elements only, but not on the multiplicities.
- `Dom::Multiset` is of the new category `Cat::Set`, and therefore offers the new features provided by this category.

`Dom::MultivariatePolynomial` – the domains of multivariate polynomials

`Dom::MultivariatePolynomial(Vars, R, ..)` creates the domain of multivariate polynomials in the variable list `Vars` over the commutative ring `R` in distributed representation.

Domain:

- `Dom::MultivariatePolynomial(<Vars <, R <, Order>>>)`

Parameters:

`Vars` — a list of indeterminates. Default: `[x,y,z]`.

`R` — a commutative ring, i.e., a domain of category `Cat::CommutativeRing`. Default: `Dom::ExpressionField(normal)`.

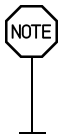
`Order` — a monomial ordering, i.e., one of the predefined orderings `LexOrder`, `DegreeOrder`, or `DegInvLexOrder` or any object of type `Dom::MonomOrdering`. Default: `LexOrder`.

Details:

- `Dom::MultivariatePolynomial` represents multivariate polynomials over arbitrary commutative rings.

  All usual algebraic and arithmetical polynomial operations are implemented, including Gröbner basis computation and some classical construction tools used in invariant theory.

- It is highly recommend to use only coefficient rings with unique zero representation. Otherwise it may happen that, e.g., a polynomial division will not terminate or a wrong degree will be returned.

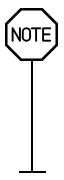


- `Dom::MultivariatePolynomial(Vars, R, Order)` creates a domain of multivariate polynomials in the variable list `Vars` over a domain `R` of category `Cat::CommutativeRing` in sparse distributed representation with respect to the monomial ordering `Order`.

- `Dom::MultivariatePolynomial()` creates a polynomial domain in the variable list `[x,y,z]` over the domain `Dom::ExpressionField(normal)` with respect to the lexicographic monomial ordering.

- `Dom::MultivariatePolynomial(Vars)` generates the polynomial domain in the variable list `Vars` over the domain `Dom::ExpressionField(normal)` with respect to the lexicographic monomial ordering is created.

- Only commutative coefficient rings of type `DOM_DOMAIN` which inherit from `Dom::BaseDomain` are allowed. If `R` is of type `DOM_DOMAIN` but does not inherit from `Dom::BaseDomain`, the domain `Dom::ExpressionField(normal)` will be used instead.



- In contrast to the domain `Dom::DistributedPolynomial`, `Dom::MultivariatePolynomia` accepts only identifiers (`DOM_IDENT`) as indeterminates. This restriction enables some further methods described below.

- Please note: For reasons of efficiency not all methods check their arguments, not even at the interactive level. In particular this is true for many access methods, converting methods and technical methods. This may cause strange error messages.

Creating Elements:

- `Dom::MultivariatePolynomial(Vars, R, Order)(p)`
- `Dom::MultivariatePolynomial(Vars, R, Order)(lm)`

Parameters:

`p` — a polynomial or a polynomial expression.
`lm` — list of monomials, which are represented as lists containing the coefficients together with the exponents or exponent vectors.

Categories:

if `Vars` has a single variable, then
`Cat::UnivariatePolynomial(R)`

else
`Cat::Polynomial(R)`

Related Domains: `Dom::DistributedPolynomial`, `Dom::Polynomial`, `Dom::UnivariatePolynomial`

Entries:

`characteristic` The characteristic of this domain.

`coeffRing` The coefficient ring of this domain as defined by the parameter `R`.

`key` The name of the domain created.

`one` The neutral element w.r.t. `"_mult"`.

`ordering` The monomial ordering defined by the parameter `Order`.

`variables` The list of variables defined by the parameter `Vars`.

`zero` The neutral element w.r.t. `"_plus"`.

Mathematical Methods

Method `borderedHessianDet`: bordered Hessian determinant of a polynomial

`borderedHessianDet(`*`dom`* `a,` *`dom`* `b <,` *`list of indeterminates`* `v>)`

- Returns the determinant of the Hessian matrix of `a` bordered by `b` with respect to `v`, which is

$$\texttt{borderedHessianDet(a,b,v)} = \det \begin{pmatrix} \frac{\partial^2 a}{\partial v_1 \partial v_1} & \cdots & \frac{\partial^2 a}{\partial v_1 \partial v_n} & \frac{\partial b}{\partial v_1} \\ \vdots & & \vdots & \vdots \\ \frac{\partial^2 a}{\partial v_n \partial v_1} & \cdots & \frac{\partial^2 a}{\partial v_n \partial v_n} & \frac{\partial b}{\partial v_n} \\ \frac{\partial b}{\partial v_1} & \cdots & \frac{\partial b}{\partial v_n} & 0 \end{pmatrix}$$

  as an element of this domain. If `v` is not given, `Vars` will be used instead.

Method `borderedHessianMat`: bordered Hessian matrix of a polynomial

`borderedHessianMat(`*`dom`* `a,` *`dom`* `b <,` *`list of indeterminates`* `v>)`

- Returns the Hessian matrix of `a` bordered by `b` with respect to `v` as an element of `Dom::Matrix(dom)`. For the definition of that matrix see method `"borderedHessianDet"`. If `v` is not given, `Vars` will be used instead.

Method `degLex`: compares two polynomials w.r.t. the graded lexicographical order

`degLex(`*`dom`* `a,` *`dom`* `b)`

- Returns `-1` if $\texttt{a} < \texttt{b}$, `0` if $\texttt{a} = \texttt{b}$, `1` if $\texttt{a} > \texttt{b}$ with respect to the graded lexicographical order, which first uses the degree order and then the lexicographical order for tie-break.

Method `degRevLex`: compares two polynomials w.r.t. the graded reverse lexicographical order

`degRevLex(`*`dom`* `a,` *`dom`* `b)`

- Returns `-1` if $\texttt{a} < \texttt{b}$, `0` if $\texttt{a} = \texttt{b}$, `1` if $\texttt{a} > \texttt{b}$ with respect to the graded reverse lexicographical order, which first uses the degree order and then the reverse lexicographical order for tie-break.

Method `hessianDet`: Hessian determinant of a polynomial

`hessianDet(`*`dom`* `a <,` *`list of indeterminates`* `v>)`

- Returns the determinant of the Hessian matrix of `a` with respect to `v`, which is

$$\texttt{hessianDet(a,v)} = \det\left(\frac{\partial^2 a}{\partial v_i \partial v_j}\right)$$

  as an element of this domain. If `v` is not given, `Vars` will be used instead.

Method `hessianMat`: Hessian matrix of a polynomial

`hessianMat(`*`dom`* `a <,` *`list of indeterminates`* `v>)`

- Returns the Hessian matrix of `a` with respect to `v`, which is

$$\texttt{hessianMat(a,v)} = \left(\frac{\partial^2 a}{\partial v_i \partial v_j}\right)$$

  as an element of `Dom::Matrix(dom)`. If `v` is not given, `Vars` will be used instead.

Method `homogeneousComponents`: list of homogeneous components of a polynomial

`homogeneousComponents(`*`dom`* `a)`

- Returns an ordered list of the homogeneous components of `a`, i.e., a list of sums of monomials with the same total degree. The list is sorted in descending total degree order.

Method `isHomogeneous`: tests if a polynomial is homogeneous

`isHomogeneous(`*`dom`* `a)`

- Returns `TRUE` if `a` is a homogeneous polynomial and `FALSE` otherwise.

Method `jacobianDet`: Jacobian determinant of a polynomial

`jacobianDet(`*`list of dom`* `ais <,` *`list of indeterminates`* `v>)`

- Returns the determinant of the Jacobian matrix of `ais`, with respect to `v` which is

$$\texttt{jacobianDet(ais,v)} = \det\left(\frac{\partial ais_i}{\partial v_j}\right)$$

  as an element of this domain. If `v` is not given, `Vars` will be used instead.

Method `jacobianMat`: Jacobian matrix of a polynomial

`jacobianMat(`*`list of dom`* `ais <, `*`list of indeterminates`* `v>)`

- Returns the Jacobian matrix of `ais`, with respect to `v` which is

$$\texttt{jacobianMat(ais,v)} = \left(\frac{\partial ais_i}{\partial v_j}\right)$$

  as an element of `Dom::Matrix(dom)`. If `v` is not given, `Vars` will be used instead.

Method `rewriteHomPoly`: rewrites a polynomial in terms of other polynomials

`rewriteHomPoly(`*`dom`* `a, `*`list of dom`* `ais, `*`list of indeterminates`* `v)`

- Computes a polynomial `g` over the ring `R` in the variable list `v`, such that `g(op(ais)) = a`, i.e., it returns the homogeneous polynomial `a` expressed in terms of the new variable list `v`, which represents the list of homogeneous polynomials `ais` respectively. For this, the sequence (order) of `ais` is used in the algorithm.
- All the polynomials `a` and `ais` must be homogeneous.
- The variables of `v` should be new variables.

Method `rewritePoly`: rewrites a polynomial in terms of other polynomials

`rewritePoly(`*`dom`* `a, `*`list of equations`* `[ai=vi] <, Unsorted>)`

- computes a polynomial `g` over the ring `R` in the variables `vi` such that `g(...,ai,...)=a`, where the `ai`'s are homogeneous polynomials of this domain, and returns the polynomial `a` expressed in terms of the new variables `vi`, or `FAIL` if this is not possible.
- This method can be used for representing a polynomial with respect to a given polynomial basis.
- When option `Unsorted` is given, the list `[ai=vi]` is not sorted. Otherwise, in a precomputation step this list will be sorted in the `ai`'s w.r.t. the graded lexicographical order (`"degLex"`).
- Please note: the algorithm depends on the order of `Vars` and `ais`.
- All the polynomials `ai` must be homogeneous.
- The variables of `vi` should be new variables.

Access Methods

Method `order`: compares two polynomials w.r.t. a given order

`order(`*`dom`* `a,` *`dom`* `b,` *`Dom::MonomOrdering`* `o)`

- Compares `a` and `b` with respect to the monomial order `o`: If $a > b$ then 1 is returned, if $a = b$ then 0 is returned and if $a < b$ then -1 is returned.

Method `sortList`: sorts a list of polynomials w.r.t. a given order

`sortList(`*`list of dom`* `ais,` *`Dom::MonomOrdering`* `o )`

- Sorts the polynomials `ais` with respect to the monomial order `o` in descending order.
- This sorting method may be not stable if `o` is not a total order.

Method `stableSort`: sorts a list of polynomials w.r.t. a given order

`stableSort(`*`list of dom`* `ais,` *`Dom::MonomOrdering`* `o )`

- Sorts the polynomials `ais` with respect to the monomial order `o` in descending order.
- This sorting method is stable, even if `o` is not a total order.

Example 1. To create the ring of multivariate polynomials in `x`, `y` and `z` over the rationals one may define

```
>> MP := Dom::MultivariatePolynomial([x, y, z], Dom::Rational)
 Dom::MultivariatePolynomial([x, y, z], Dom::Rational, LexOrder)
```

The elementary symmetric polynomials of this domain are

```
>> s1 := MP(x + y + z)
                              x + y + z
>> s2 := MP(x*y + x*z + y*z)
                           x y + x z + y z
>> s3:=MP(x*y*z)
                                 x y z
```

A polynomial is called symmetric if it remains unchanged under every possible permutation of variables as, e.g.:

```
>> s3=s3(MP(y), MP(z), MP(x))

                          x y z = x y z
```

These polynomials arise naturally in studying the roots of a polynomial. To show this, we first have to create an univariate polynomial, e.g., in `U` over `MP`, and generate a polynomial in `U` with roots in `x`, `y` and `z`.

```
>> UP:=Dom::UnivariatePolynomial(U, MP)

 Dom::UnivariatePolynomial(U, Dom::MultivariatePolynomial(

    [x, y, z], Dom::Rational, LexOrder), LexOrder)

>> f := UP((U - x)*(U - y)*(U - z))

        3                   2
       U  + (- x - y - z) U  + (x y + x z + y z) U - x y z

>> UP(U^3)-s1*UP(U^2)+s2*UP(U)+(-1)^3*s3

        3                   2
       U  + (- x - y - z) U  + (x y + x z + y z) U - x y z
```

This exemplifies that the coefficients of `f` are (elementary) symmetric polynomials in its roots.

From the fundamental theorem of symmetric polynomials we know that every symmetric polynomial can be written uniquely as a polynomial in the elementary symmetric polynomials. Thus we can rewrite the following symmetric polynomial `s` in the elementary symmetric polynomials `s1`, `s2` and `s3`,

```
>> s:=MP(x^3*y+x^3*z+x*y^3+x*z^3+y^3*z+y*z^3)

             3      3        3      3    3        3
            x  y + x  z + x y  + x z  + y  z + y z

>> S:=MP::rewritePoly(s,[s1=S1,s2=S2,s3=S3])

                      2                   2
                     S1  S2 - S1 S3 - 2 S2
```

where these polynomials are represented by the three new variables `S1`, `S2` and `S3` respectively. To see that this new polynomial `S` in the new variables indeed represents the old original polynomial `s`, we simply have to plug in the three elementary symmetric polynomials into `S`:

```
>> S(s1,s2,s3,Expr)

             3      3        3      3    3        3
            x  y + x  z + x y  + x z  + y  z + y z
```

When one has a given list of polynomials, e.g., like:

```
>> l:=[3*s1,2*s1,s1,s3]

      [3 x + 3 y + 3 z, 2 x + 2 y + 2 z, x + y + z, x y z]
```

and one wants to sort them in an appropriate order, one may use one of the following two methods.

```
>> MP::sortList(l,Dom::MonomOrdering(DegLex(3)))

      [x y z, 2 x + 2 y + 2 z, x + y + z, 3 x + 3 y + 3 z]

>> MP::stableSort(l,Dom::MonomOrdering(DegLex(3)))

      [x y z, 3 x + 3 y + 3 z, 2 x + 2 y + 2 z, x + y + z]
```

In the first sorted list the order of the three polynomials of the same degree has changed, while with the second method this order remains stable.

Example 2. Let $G \subseteq \mathrm{GL}(n,k)$ be a finite (matrix) subgroup of the general linear group. Then a polynomial $f \in k[x_1,\ldots,x_n]$ is called *invariant under G*, if for all $A \in G$

$$f(\mathbf{x}) = f(A \cdot \mathbf{x})$$

where $\mathbf{x} = (x_1,\ldots,x_n)$.
The symmetric polynomials `s1`, `s2` and `s3` from the previous example are invariants under the symmetric group S_3. In fact, these three fundamental invariants yet generate the whole ring of invariants of S_3.

Now let us examine the invariants of the famous icosahedral group. One may find a representation of this group in *H. F. Blichfeldt: Finite collineation groups, University of Chicago Press, 1917.* on page 73.

$$S' = \begin{pmatrix} \epsilon^3 & 0 \\ 0 & \epsilon^2 \end{pmatrix}, \quad U' = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}, \quad T' = \begin{pmatrix} \alpha & \beta \\ \beta & -\alpha \end{pmatrix}, \ \epsilon^5 = 1, \ \alpha = \frac{\epsilon^4 - \epsilon}{\sqrt{5}}, \ \beta = \frac{\epsilon^2 - \epsilon^3}{\sqrt{5}}$$

The group is generated from these three matrices, has 120 elements and is thus a finite subgroup, even of the special linear group $\mathrm{SL}(2,\mathbb{Q}(\epsilon))$. It is also well known that

$$I_1 = x_1 {x_2}^{11} - 11 {x_1}^6 {x_2}^6 - {x_1}^{11} x_2$$

is a fundamental invariant of degree 12 of this group. To declare I_1 in MuPAD one has first to define the polynomial domain.

```
>> MP:=Dom::MultivariatePolynomial([x1,x2],Dom::Rational)

 Dom::MultivariatePolynomial([x1, x2], Dom::Rational, LexOrder)

>> i1:=MP(x1*x2^(11)-11*x1^6*x2^6-x1^(11)*x2)
```

```
                    11           6   6        11
                - x1   x2 - 11 x1  x2  + x1 x2
```

From the invariant I_1 one can compute a further fundamental invariant I_2 with

```
>> i2:=MP::hessianDet(i1)
```

```
         20           15   5           10   10
 - 121 x1   + 27588 x1   x2  - 59774 x1   x2   -

            5   15         20
    27588 x1  x2   - 121 x2
```

But to get more simple coefficients we choose I_2 as

```
>> i2:=-1/121*MP::hessianDet(i1)
```

```
      20         15   5         10   10         5   15     20
    x1   - 228 x1   x2  + 494 x1   x2   + 228 x1  x2   + x2
```

instead. Similar we obtain a third fundamental invariant I_3 with

```
>> i3:=1/20*MP::jacobianDet([i1,i2])
```

```
   30         25   5           20   10           10   20
 x1   + 522 x1   x2  - 10005 x1   x2   - 10005 x1   x2   -


          5   25     30
    522 x1  x2   + x2
```

In contrast to the symmetric groups, where all invariants can be uniquely represented by the fundamental invariants, the fundamental invariants of this group have an algebraic relation, a so-called syzygy between them. It is possible to represent ${I_3}^2$ in two ways:

```
>> MP::rewritePoly(i3^2,[i1=I1,i2=I2,i3=I3])
```

```
                                5     3
                        - 1728 I1  + I2
```

```
>> MP::rewritePoly(i3^2,[i1=I1,i2=I2,i3=I3],Unsorted)
```

```
                                2
                               I3
```

And hence we get the syzygy:

```
>> MP::rewritePoly(i3^2,[i1=I1,i2=I2,i3=I3],Unsorted)-
   MP::rewritePoly(i3^2,[i1=I1,i2=I2,i3=I3]) = 0
```

```
                           5     3     2
                    1728 I1  - I2  + I3  = 0
```

Super-Domain: `Dom::DistributedPolynomial`

Axioms

if R has `Ax::normalRep`, then
`Ax::normalRep`

if R has `Ax::canonicalRep`, then
`Ax::canonicalRep`

Background:

- The algorithms used for rewriting polynomials, i.e., for expressing polynomials in terms of some other homogeneous polynomials stem from

  - Winfried Fakler. Algorithmen zur symbolischen Lösung homogener linearer Differentialgleichungen. Diplomarbeit, Universität Karlsruhe (1994).

Changes:

- `Dom::MultivariatePolynomial` is a new domain.

`Dom::Numerical` – the field of numbers

`Dom::Numerical` is the field of numbers.

Details:

- `Dom::Numerical` is the domain of numbers represented by one of the kernel domains `DOM_INT`, `DOM_RAT`, `DOM_FLOAT`, or `DOM_COMPLEX`.

- `Dom::Numerical` is of category `Cat::Field` due to pragmatism. This domain actually is not a field because `bool(1.0 = float(3) / float(3))` returns `FALSE`, for example.

- Elements of `Dom::Numerical` are usually not created explicitly. However, if one creates elements using the usual syntax, it is checked whether the input expression can be converted into a number (see below).

  This means that `Dom::Numerical` is a façade domain which creates elements of domain type `DOM_INT`, `DOM_RAT`, `DOM_FLOAT` or `DOM_COMPLEX`. Every system function dealing with numbers can be applied, and computations in this domain are performed efficiently.

- `Dom::Numerical` has no normal representation, because `0` and `0.0` both represent zero.

- Viewed as a differential ring, `Dom::Numerical` is trivial. It only contains constants.

Creating Elements:

- `Dom::Numerical(x)`

Parameters:

`x` — an arithmetical expression

Categories:

`Cat::DifferentialRing`, `Cat::Field`

Related Domains: `Dom::Complex`, `Dom::Float`, `Dom::Integer`, `Dom::Rational`, `Dom::Real`

Details:

- If `x` is a constant arithmetical expression such as `sin(2)` or `PI + 2`, the system function `float` is applied to convert `x` into a floating point approximation.

  An error message is issued if the result of this conversion is not of domain type `DOM_FLOAT` or `DOM_COMPLEX`.

Entries:

`characteristic` is zero.

Mathematical Methods

Method `D`: the differential operator for numbers

`D(`*`dom`* `a)`

- This method returns the derivative of `a`, which is zero.
- See the function `D` for details and further calling sequences.

Method `diff`: differentiation of numbers

`diff(`*`dom`* `a, variable x)`

- This method differentiates `a` with respect to `x`, which results in zero.
- See the function `diff` for details and further calling sequences.

Method `norm`: the absolute value of numbers

```
norm(dom a)
```

- This method returns $|a|$.

Method `random`: random number generation

```
random()
```

- This methods returns a randomly generated number. The real and imaginary part of this number are generated as $\tan(r)$, where r is uniformly distributed in the interval $]-\pi/2, \pi/2[$.

Conversion Methods

Method `convert`: conversion of objects into numbers

```
convert(any x)
```

- This method tries to convert `x` into a number of type `Dom::Numerical` and returns `FAIL` if this is not possible.
- If `x` is of the domain type `DOM_INT`, `DOM_RAT`, `DOM_FLOAT` or `DOM_COMPLEX`, `x` is returned.
  Otherwise `float(x)` is computed and the result is returned, if it is of the domain type `DOM_FLOAT` or `DOM_COMPLEX`. If it is not, `FAIL` is returned.

Method `convert_to`: conversion into other domains

```
convert_to(dom a, any T)
```

- This method tries to convert the number `a` into an element of the domain `T`.
- If the conversion fails, `FAIL` is returned.
- It currently handles the following domains for `T`: `DOM_INT`, `Dom::Integer`, `DOM_RAT`, `Dom::Rational`, `DOM_FLOAT`, `Dom::Float` and `DOM_COMPLEX`.

Method `testtype`: type checking

```
testtype(any a, dom T)
```

- This method checks whether it can convert `a` into the domain `Dom::Numerical`. The method returns `TRUE` if it can perform the conversion, and `FAIL` otherwise.
- This method is called from the function `testtype`.

Example 1. `Dom::Numerical` contains numbers of the domains `DOM_INT`, `DOM_RAT`, `DOM_FLOAT` and `DOM_COMPLEX`:

```
>> Dom::Numerical(2), Dom::Numerical(2/3),
   Dom::Numerical(3.141), Dom::Numerical(2 + 3*I)

                    2, 2/3, 3.141, 2 + 3 I
```

Constant arithmetical expressions are converted into a real and complex floating point number, respectively, i.e., into an element of the domain `DOM_FLOAT` or `DOM_COMPLEX` (see the function `float` for details):

```
>> Dom::Numerical(exp(5)), Dom::Numerical(sin(2/3*I) + 3)

                148.4131591, 3.0 + 0.717158461 I
```

Note that the elements of this domain are elements of kernel domains, there are no elements of the domain type `Dom::Numerical`!

An error message is issued for non-constant arithmetical expressions:

```
>> Dom::Numerical(sin(x))

 Error: illegal arguments [Dom::Numerical::new]
```

Example 2. `Dom::Numerical` is regarded as a field, and it therefore can be used as a coefficient ring of polynomials or as a component ring of matrices, for example.

We create the domain of matrices of arbitrary size (see `Dom::Matrix`) with numerical components:

```
>> MatN := Dom::Matrix(Dom::Numerical)

                  Dom::Matrix(Dom::Numerical)
```

Next we create a banded matrix, such as:

```
>> A := MatN(4, 4, [-PI, 0, PI], Banded)

   +-                                                    -
+
   |        0,       3.141592654,       0,           0       |
   |                                                         |
   |  -3.141592654,       0,       3.141592654,      0       |
   |                                                         |
   |        0,      -3.141592654,       0,      3.141592654  |
   |                                                         |
   |        0,            0,      -3.141592654,      0       |
   +-                                                    -
+
```

and a row vector with four components as a 1×4 matrix:

```
>> v := MatN([[2, 3, -1, 0]])
```

```
                          +-            -+
                          | 2, 3, -1, 0 |
                          +-            -+
```

Vector-matrix multiplication can be performed with the standard operator `*` for multiplication:

```
>> v * A
```

```
    +-                                                          -
+
    | -9.424777961, 9.424777961, 9.424777961, -3.141592654 |
    +-                                                          -
+
```

Finally we compute the determinant of the matrix `A`, using the function `linalg::det` of the `linalg` package:

```
>> linalg::det(A)
```

```
                              97.40909104
```

Super-Domain: `Dom::ArithmeticalExpression`

Axioms

`Ax::canonicalRep`, `Ax::systemRep`,
`Ax::efficientOperation("_divide")`,
`Ax::efficientOperation("_mult")`,
`Ax::efficientOperation("_invert")`

Changes:

- No changes.

`Dom::PermutationGroup` – permutation groups

`Dom::PermutationGroup(n)` creates the domain of all permutations of n elements.

Domain:

- `Dom::PermutationGroup(n)`

Parameters:

`n` — positive integer

Details:

- A permutation of n elements is a bijective mapping of the set $\{1, \ldots, n\}$ onto itself.

The domain element `Dom::PermutationGroup(n)(l)` represents the bijective mapping of the first n positive integers that maps the integer i to `l[i]`, for $1 \leq i \leq n$.

Creating Elements:

- `Dom::PermutationGroup(n)(l)`

Parameters:

`l` — list or array consisting of the first n integers in some order.

Categories:

`Cat::Group`

Related Domains: `Dom::DihedralGroup`

Entries:

`one`	the identical mapping of the set $\{1, \ldots, n\}$ to itself.

Mathematical Methods

Method `_mult`: product of permutations

`_mult(`*dom* `a1, ...)`

- The product `a1*a2* ... *ak` of permutations is defined to be the mapping that assigns, to every integer `i` between 1 and n, the integer `a1(a2( ... ak(i) ... ))`.
- This method overloads the function `_mult`.

Method `_invert`: inverse of a permutation

```
_invert(dom a)
```

- This method computes a permutation `b` such that `a*b` is the identity mapping.
- This method overloads the function `_invert`.

Method `func_call`: function value of a permutation at a point

```
func_call(dom a, integer i)
```

- This method overloads the round brackets `( ... )`, i.e. it may be called in the form `a(i)`.
- It computes the function value of `a` at `i`, i.e., the integer that `i` is mapped to by the permutation `a`; `i` must be an integer between 1 and n.

Method `cycles`: cycle representation of a permutation

```
cycles(dom a)
```

- This method computes a cycle representation of `a`. A cycle representation is a list `[orbit1, ..., orbitk]`; each of the orbits is a list of integers of the form `[i, a(i), a(a(i)), ... ]` with just as many elements such that `i` does not occur in it for a second time; and each integer between 1 and n appears in exactly one of the orbits.

Method `order`: order of a permutation

```
order(dom a)
```

- The order of `a` is defined to be the least positive integer `k` for which a^k is the identity.

Method `inversions`: number of inversions

```
inversions(dom a)
```

- This method computes the number of all pairs i, j of integers for which $i < j$ but $a(i) > a(j)$.

Method `random`: random permutation

```
random()
```

- This method returns a random element of the permutation group.

Conversion Methods

Method `convert`: conversion of an object into a permutation

```
convert(any x)
```

- This method tries to convert `x` into a permutation. This is only possible if `x` is a list or an array in which each of the integers 1 through n occurs exactly once.

Method `convert_to`: conversion of a permutation into another type

```
convert_to(dom a, any T)
```

- Tries to convert `a` into type `T`. Currently, only a conversion into a list of type `DOM_LIST` is possible.

Method `expr`: convert a permutation into a list

```
expr(dom a)
```

- This method returns a list such that generating a permutation from that list would result in `a`.

Example 1. Consider the group of permutations of the first seven positive integers:

```
>> G:=Dom::PermutationGroup(7)

                    Dom::PermutationGroup(7)
```

We enter an element by providing the image of 1, 2, etc. under the permutation.

```
>> a:=G([2,4,6,1,3,5,7])

                      [2, 4, 6, 1, 3, 5, 7]

>> a(3)

                                  6
```

Super-Domain: `Dom::BaseDomain`

Axioms

```
Ax::canonicalRep
```

Changes:

- No changes.

`Dom::Polynomial` – the domains of polynomials in arbitrarily many indeterminates

`Dom::Polynomial(R, ..)` creates the domain of polynomials in arbitrarily many indeterminates over the commutative ring `R` in distributed representation.

Domain:

- `Dom::Polynomial(<R <, Order>>)`

Parameters:

`R`	—	a commutative ring, i.e., a domain of category `Cat::CommutativeRing`. Default: `Dom::ExpressionField(normal)`.
`Order`	—	a monomial ordering, i.e., one of the predefined orderings `LexOrder`, `DegreeOrder`, or `DegInvLexOrder` or an element of the domain `Dom::MonomOrdering`. Default: `LexOrder`.

Details:

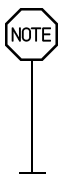
- `Dom::Polynomial` represents polynomials in arbitrarily many indeterminates over arbitrary commutative rings.

  It is simply a frontend to the domain `Dom::DistributedPolynomial([],R,Order)` and thus all usual algebraic and arithmetical polynomial operations are implemented. Please see the documentation for `Dom::DistributedPolynomial` for a list of methods.

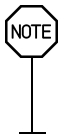
- `Dom::Polynomial(R, Order)` creates a domain of polynomials in arbitrarily many indeterminates over a domain of category `Cat::CommutativeRing` in sparse distributed representation with respect to the monomial ordering `Order`.

- If `Dom::Polynomial` is called without any argument, a polynomial domain over the domain `Dom::ExpressionField(normal)` with respect to the lexicographic monomial ordering is created.

- Only commutative coefficient rings of type `DOM_DOMAIN` which inherit from `Dom::BaseDomain` are allowed. If `R` is of type `DOM_DOMAIN` but does not inherit from `Dom::BaseDomain`, the domain `Dom::ExpressionField(normal)` will be used instead.



- Only identifiers should be used as polynomial indeterminates, since when creating a new element from a polynomial or a polynomial expression the function `indets` is first called to get the identifiers and then the polynomial is created with respect to these identifiers.
- It is highly recommend to use only coefficient rings with unique zero representation. Otherwise it may happen that, e.g., a polynomial division will not terminate or a wrong degree will be returned.



- Please note that for reasons of efficiency not all methods check their arguments, not even at the interactive level. In particular, this is true for many access methods, converting methods and technical methods. Thus, improper use of these methods may result in confusing error messages.

Creating Elements:

- `Dom::Polynomial(R<, Order>)(p)`
- `Dom::Polynomial(R<, Order>)(lm,v)`

Parameters:

`p` — a polynomial or a polynomial expression.
`lm` — list of monomials, which are represented as lists containing the coefficients together with the exponents or exponent vectors.
`v` — list of indeterminates.

Categories:

`Cat::Polynomial(R)`

Related Domains: `Dom::DistributedPolynomial`, `Dom::MultivariatePolynomial`, `Dom::UnivariatePolynomial`

Entries:

`characteristic` The characteristic of this domain, which is the characteristic of `R`.

`coeffRing` The coefficient ring of this domain as defined by the parameter `R`.

`key` The name of the domain created.

`one` The neutral element w.r.t. `"_mult"`, which is `R::one`.

`ordering` The monomial order as defined by the parameter `Order`.

`zero` The neutral element w.r.t. `"_plus"`, which is `R::zero`.

Example 1. The following call creates the polynomial domain over the rationals.

```
>> PR:=Dom::Polynomial(Dom::Rational)

            Dom::Polynomial(Dom::Rational, LexOrder)
```

Since the monomial ordering was not specified, this domain is created with the default value for this parameter.

It is rather easy to create elements of this domain, as, e.g.,

```
>> a := PR(x*(2*x + y^3) - 7/2)

                           2      3
                       2 x  + x y  - 7/2

>> b := PR(x*(2*t + z^3) - 6)

                                   3
                       2 t x + x z  - 6

>> c := a^2-b/3+3

                4      3  3    2  6       2        3
 - 2/3 t x + 4 x  + 4 x  y  + x  y  - 14 x  - 7 x y  -

          3
    1/3 x z  + 69/4
```

Super-Domain: `Dom::DistributedPolynomial`

Axioms

if R has `Ax::normalRep`, then
`Ax::normalRep`

if R has `Ax::canonicalRep`, then
`Ax::canonicalRep`

Background:

- To create polynomials from expressions with no suitable indeterminates the dummy variable `_dummy` is introduced. With this variable it is possible to create elements from constants which otherwise would fail. The drawback of this approach is that two mathematically equal polynomials may have variable lists which differ by this dummy variable.

Changes:

- `Dom::Polynomial` used to be `Dom::PolynomialExplicit([],R,Order)`.
- The former implementation of this domain does no longer exist and has been completely reimplemented.
- It is now allowed to call this domain with no, one or two arguments. With the second argument one can now choose an appropriate monomial ordering.
- The method `"indets"` now returns a set of indeterminates.
- The methods `"Rep"`, `"decompose"`, `"int"`, `"makeIntegral"`, `"monic"`, `"orderedVariableList"`, `"ordering"`, `"reductum"`, `"resultant"` were added.

`Dom::Product` – homogeneous direct products

`Dom::Product(Set, n)` is an n-fold direct product of the domain *Set*.

Domain:

- `Dom::Product(Set <, n>)`

Parameters:

`Set` — an arbitrary domain of elements, i.e., a domain of category `Cat::BaseCategory`
`n` — the dimension of the product (a positive integer); default is 1

Creating Elements:

- `Dom::Product(Set<, n>)(e1, e2, ..., en)`
- `Dom::Product(Set<, n>)(List)`

Parameters:

`e1, e2, ..., en` — elements of `Set` or objects convertible into such
`List` — a list of `n` elements of `Set` or objects convertible into such

Categories:

`Cat::HomogeneousFiniteProduct(Set)`

Details:

- `Dom::Product(Set, n)(e1, e2, ..., en)` creates the n-tuple $(e_1, e_2, \ldots, e_n)$.

  The objects `e1`, `e2, ...`, `en` must be convertible into elements of the domain `Set`, otherwise an error message is issued.

- `Dom::Product(Set, n)(List)` creates the n-tuple $(l_1, l_2, \ldots, l_n)$.

  The n elements l_i of `List` must be convertible into elements of the domain `Set`, otherwise an error message is issued.

  The list must consist of exactly `n` elements, otherwise an error message is issued.

- Following to the definition of a direct product many of the methods such as `"D"` and `"_negate"` just map the operation to all the components of the tuple.

  Most n-ary methods like `"_plus"` and `"_mult"` apply the operation component-wise to the tuples.

Entries:

`card` is the cardinal number of `Dom::Product(Set, n)`, which is equal to `n`.

`coeffRing` is the domain `S`.

`one` is the n-tuple `(Set::one, Set::one, ..., Set::one)`. This entry only exists if `Set` is a monoid, i.e., a domain of category `Cat::Monoid`.

`zero` is the n-tuple `(Set::zero, Set::zero, ..., Set::zero)`. This entry only exists if `Set` is an Abelian group, i.e., a domain of category `Cat::AbelianGroup`.

Mathematical Methods

Method `_divide`: divide tuples

`_divide(`*`dom`* `x, `*`dom`* `y)`

- This method divides the tuple `x` by `y`, i.e., it divides the ith component of `x` by the ith component of `y` (i ranges from 1 to n).
- This method only exists if `Set` is a (multiplicative) group, i.e., a domain of category `Cat::Group`.
- This method overloads the function `_divide` for n-tuples, i.e., one may use it in the form `x / y`, or in functional notation: `_divide(x, y)`.

Method `_invert`: computes the inverse of a tuple

```
_invert(dom x)
```

- The inverse of a tuple is the inverse of each component of `x`.
- This method only exists if `Set` is a (multiplicative) group, i.e., a domain of category `Cat::Group`.
- This method overloads the function `_invert` for n-tuples, i.e., one may use it in the form `1/x` or `x^(-1)`, or in functional notation: `_inverse(x)`.

Method `_less`: less-than relation

```
_less(dom x, dom y)
```

- returns `TRUE` if `x` is lexically smaller than `y`.
- An implementation is provided only if `Set` is an ordered set, i.e., a domain of category `Cat::OrderedSet`.
- This method overloads the function `_less` for n-tuples, i.e., one may use it in the form `x < y`, or in functional notation: `_less(x, y)`.

Method `_mult`: multiplies tuples by tuples and scalars

```
_mult(any x, any y, ...)
```

- If `x` and `y` both are n-tuples defined over `Set` the n-tuple with the ith component defined by $x_i \cdot y_i$ (i ranges from 1 to n) is returned.
- If `x` is not of the type `Dom::Product(Set,n)`, it is considered as a scalar which is multiplied to each component of the n-tuple `y` (and vice versa).
- This method only exists if `Set` is a semigroup, i.e., a domain of category `Cat::SemiGroup`.
- This method also handles more than two arguments. In this case, the argument list is split into two parts of the same length which both are multiplied with the function `_mult`. These two result are multiplied again with `_mult` whose result then is returned.
- This method overloads the function `_mult` for n-tuples, i.e., one may use it in the form `x * y`, or in functional notation: `_mult(x, y)`.

Method `_negate`: negates an n-tuple

```
_negate(dom x)
```

- The negative of an n-tuple is the negative of each of its components.
- This method overloads the function `_negate` for n-tuples, i.e., one may use it in the form `-x`, or in functional notation: `_negate(x)`.

Method `_power`: the ith power of a tuple

```
_power(dom x, integer i)
```

- This method raises each component of `x` to the ith power.
- An implementation is provided only if `Set` is a semigroup, i.e., a domain of category `Cat::SemiGroup`.
- This method overloads the function `_power` for n-tuples, i.e., one may use it in the form `x^i`, or in functional notation: `_power(x, i)`.

Method `_plus`: adds tuples

```
_plus(dom x, dom y, ...)
```

- Returns the sum $x + y + \cdots$.
- The sum of two n-tuples x and y is defined component-wise as $(x_1 + y_1, \ldots, x_n + y_n)$.
- This method overloads the function `_plus` for n-tuples, i.e., one may use it in the form `x + y`, or in functional notation: `_plus(x, y)`.

Method `D`: the differential operator

```
D(dom x)
```

- This method returns the n-tuple which results from taking the derivative of each component of `x` using the method `"D"` of the domain `S`.
- An implementation is provided only if `Set` is a partial differential ring, i.e., a domain of category `Cat::PartialDifferentialRing`.
- This method overloads the operator `D` for n-tuples, i.e., one may use it in the form `D(x)`.

Method `diff`: differentiation of n-tuples

```
diff(dom a, variable x)
```

- This method returns the n-tuples which results from differentiating each component of `a` using the method `"diff"` of the domain `S`.
- This method overloads the function `diff` for n-tuples, i.e., one may use it in the form `diff(a, x)`.
- An implementation is provided only if `Set` is a partial differential ring, i.e., a domain of category `Cat::PartialDifferentialRing`.

Method `equal`: test on equality of n-tuples

```
equal(dom x, dom y)
```

- This method tests if the tuple `x` is equal to `y`, and returns `TRUE`, `FALSE` or `UNKNOWN`, respectively. `x` and `y` are equal if and only if for each index i from 1 to n the ith components of `x` and `y` are equal.

Method `intmult`: multiple of a tuple

```
intmult(dom x, integer k)
```

- The k-multiple of an n-tuple is the tuple consisting of the k-multiples of its components (which are calculated by the method `"intmult"` of `Set`).
- An implementation is provided only if `Set` is an Abelian semigroup, i.e., a domain of category `Cat::AbelianSemiGroup`.

Method `iszero`: test on zero

```
iszero(dom x)
```

- This method checks whether the tuple `x` consists of zero components only and returns `TRUE`, `FALSE` and `UNKNOWN`, respectively.
- Note that there may be more than one representation of the zero n-tuple if `R` does not have `Ax::canonicalRep`.
- This method overloads the function `iszero` for n-tuples, i.e., one may use it in the form `iszero(x)`.

Method `random`: random tuple generation

```
random()
```

- This method returns a random n-tuple. It uses the method `"random"` of the domain `Set` to randomly generate the components of the tuple.

Access Methods

Method `_index`: tuple indexing

```
_index(dom x, index i)
```

- Returns the ith component of the tuple `x` (an error message is issued if `i` is less than one or greater than the number of components of `x`).
- See also the method `"op"`.
- This method overloads the function `_index` for n-tuples, i.e., one may use it in the form `x[i]`, or in functional notation: `_index(x, i)`.

Method `map`: apply a function to tuple components

`map(`*`dom`* `x, `*`function`* `func<, `*`any`* `expr, ...>)`

- This method maps the function `func` onto the components of the n-tuple `x`, with the additional function parameters `expr, ...` passed to `func`, if given.

  See the system function `map` for details.
- Note that the function values will *not* be implicitly converted into elements of the domain `Set`. One has to take care that the function calls return elements of the domain type `Set`.

- This method overloads the function `map` for n-tuples, i.e., one may use it in the form `map(x, func, ...)`.

Method `mapCanFail`: apply a function to tuple components

`mapCanFail(`*`dom`* `x, `*`function`* `func<, `*`any`* `expr, ...>)`

- This method works like the method `"map"` but returns `FAIL` if one of the function calls returned `FAIL`.

Method `op`: components of a tuple

`op(`*`dom`* `x, `*`positive integer`* `i)`

- Returns the ith component of the tuple `x`, or `FAIL` if `i` is greater than the number of components of `x`.
- See also the method `"_index"`.
- This method overloads the function `op` for n-tuples, i.e., one may use it in the form `op(x, i)`.

`op(`*`dom`* `x)`

- Returns a sequence of all components of `x`.

Method `set_index`: assigning tuple components

`set_index(`*`dom`* `x, `*`index`* `i, `*`Set`* `e)`

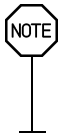
- Replaces the ith component of the tuple `x` by `e`.
- See also the method `"subsop"`.
- This method does not check whether `e` has the correct type.

- This method overloads the indexed assignment `_assign` for n-tuples, i.e., one may use it in the form `x[i] := e`, or in functional notation: `_assign(x[i], e)`.

Method `subs`: substitution of tuple components

```
subs(dom x, ...)
```

- This method applies the function `subs` with additionally given parameters to the entries of the tuple `x`.
- The objects obtained by the substitutions will not be implicitly converted into elements of the domain `Set`. One has to take care that the substitutions return elements of the domain `Set`.



- This method overloads the function `subs` for n-tuples, i.e., one may use it in the form `subs(x, ...)`. See `subs` for details and calling sequences.

Method `testEach`: check every component for a certain condition

```
testEach(dom x, function func<, any expr, ...>)
```

- This method maps the function `func` onto the components of the tuple `x`, with the additional function parameters `expr, ...` passed to `func`, if given.
  It returns `TRUE` if all of these function calls returned `TRUE`, and `FALSE` otherwise.
- `func` must return either `TRUE` or `FALSE`, otherwise a runtime error is raised.

Method `testOne`: check an component for a certain condition

```
testOne(dom x, function func<, any expr, ...>)
```

- This method maps the function `func` onto the components of the tuple `x`, with the additional function parameters `expr, ...` passed to `func`, if given.
  It returns `TRUE` if one of these function calls returned `TRUE`, and `FALSE` otherwise.
- `func` must return either `TRUE` or `FALSE`, otherwise a runtime error is raised.

Method `zip`: combine tuples component-wise

```
zip(dom x, y, function func<, any expr, ...>)
```

- This method combines the tuples `x` and `y` component-wise, where the function call `func(a, b<, expr, ...>)` is executed for each pair (x_i, y_i) of tuple components x_i of `x` and y_i of `y`.
  See the system function `zip` for details.

- The function values will not be implicitly converted into elements of the domain `Set`. One has to take care that the function calls return elements of the domain `Set`.

- This method overloads the function `zip` for n-tuples, i.e., one may use it in the form `zip(x, y, func, ...)`.

Method `zipCanFail`: combine tuples component-wise

`zipCanFail(`*`dom`* `x, y,` *`function`* `func<,` *`any`* `expr, ...>)`

- This method works like the method `"zip"` but returns `FAIL` if one of the function calls return `FAIL`.

Conversion Methods

Method `convert`: conversion into an n-tuple

`convert(`*`list`* `List)`

- Tries to convert `List` into an element of the domain `Dom::Product(Set, n)`. This can be done if `List` is a list of `n` elements where each element can be converted into an element of the domain `Set`.
- `FAIL` is returned if this conversion fails.

`convert(`*`any`* `e1<,` *`any`* `e2, ...>)`

- Tries to convert the arguments into an element of the domain `Dom::Product(Set, n)`. This can be done if exactly `n` arguments are given where each argument can be converted into an element of the domain `Set`.
- `FAIL` is returned if this conversion fails.

Method `expr`: conversion into an object of a kernel domain

`expr(`*`dom`* `x)`

- This method maps the method `"expr"` of `Set` to each element of `x` and returns a list of the converted elements, i.e., an object of type `DOM_LIST`.
- This method overloads the function `expr` for n-tuples, i.e., one may use it in the form `expr(x)`.

Example 1. Define the 3-fold direct product of the rational numbers:

```
>> P3 := Dom::Product(Dom::Rational, 3)
                  Dom::Product(Dom::Rational, 3)
```

and create elements:

```
>> a := P3([1, 2/3, 0])
                              [1, 2/3, 0]
>> b := P3(2/3, 4, 1/2)
                             [2/3, 4, 1/2]
```

We use the standard arithmetical operators to calculate with such tuples:

```
>> a + b, a*b, 2*a
          [5/3, 14/3, 1/2], [2/3, 8/3, 0], [2, 4/3, 0]
```

Some system functions were overloaded for such elements, such as `diff`, `map` or `zip` (see the description of the corresponding methods `"diff"`, `"map"` and `"zip"` above).

For example, to divide each component of `a` by 2 we enter:

```
>> map(a, '/', 2)
                             [1/2, 1/3, 0]
```

The quoted character `'/'` is another notation for the function `_divide`, the functional form of the division operator `/`.

Be careful that the mapping function returns elements of the domain the product is defined over. This is not checked by the function `map` (for efficiency reasons) and may lead to "invalid" tuples. For example:

```
>> b := map(a, sin); domtype(b)
                      [sin(1), sin(2/3), 0]

                  Dom::Product(Dom::Rational, 3)
```

But the components of `b` are no longer rational numbers!

Super-Domain: `Dom::BaseDomain`

Axioms

if `Set` has `Ax::canonicalRep`, then
`Ax::canonicalRep`

if `Set` has `Cat::AbelianMonoid` and `Set` has `Ax::normalRep`, then
`Ax::normalRep`

Changes:

- New entry `coeffRing`.

`Dom::Quaternion` – the skew field of quaternions

The domain `Dom::Quaternion` represents the skew field of quaternions.

Creating Elements:

- `Dom::Quaternion(listi)`
- `Dom::Quaternion(ex)`
- `Dom::Quaternion(M)`

Parameters:

`listi` — a list containing four elements of type `Type::Real`
`ex` — arithmetical expression
`M` — A matrix of type `Dom::Matrix(Dom::Complex)`. It has to be of a special form described in the Details section.

Categories:

`Cat::SkewField`

Related Domains: `Dom::Complex`

Details:

- Quaternions are usually defined to be complex 2×2–matrices of the special form

$$\begin{pmatrix} a+bI & -c-dI \\ c-dI & a-bI \end{pmatrix},$$

  where a, b, c, d are real numbers. Another usual notation is $a + bi + cj + dk$; the subfield of those quaternions for which $c = d = 0$ is isomorphic to the field of complex numbers.

- The domain `Dom::Quaternion` regards these fields as being identical, and it allows both notations that have been mentioned, as well as simply `[a,b,c,d]`.

- If you enter a quaternion as an arithmetical expression `ex`, the identifiers `i`, `j`, and `k` are understood in the way mentioned above; `I`, `J`, and `K` may be used alternatively, and you may also mix small and capital letters. Every subexpression of `ex` not containing one of these must be real and constant.

  Be sure that you have not assigned a value to one of the identifiers mentioned. 

- `Dom::Quaternion` has the domain `Dom::BaseDomain` as its super domain, i.e., it inherits each method which is defined by `Dom::BaseDomain` and not re-implemented by `Dom::Quaternion`. Methods described below are re-implemented by `Dom::Quaternion`.

Entries:

`characteristic`	the characteristic of this domain is 0
`one`	the unit element; it equals `Dom::Quaternion([1,0,0,0])`.
`size`	the number of quaternions is `infinity`.
`zero`	The zero element; it equals `Dom::Quaternion([0,0,0,0])`.

Mathematical Methods

Method `_mult`: multiplies quaternions

```
_mult(dom x , dom y, ...)
```

- This method overloads `_mult`.
- Returns the product $xy\cdots$ of quaternions.

Method `_plus`: adds quaternions

```
_plus(dom x , dom y, ...)
```

- This method overloads `_plus`.
- Returns the sum $x + y + \cdots$ of quaternions.

Method `_power`: the n-th power of a quaternion

```
_power(dom x, rational n)
```

- This method overloads `_power`.
- This method computes x^n for rational numbers `n`.

Method `Im`: returns the imaginary (vectorial) part of a quaternion.

`Im(`*`dom`* `x)`

- This method overloads `Im`.
- Returns the imaginary (vectorial) part of $x := a + bi + cj + dk$, this is $bi + cj + dk$.
- The result is still a quaternion.

Method `Re`: returns the real part of a quaternion.

`Re(`*`dom`* `x)`

- This method overloads `Re`.
- Returns the real part of $x := a + bi + cj + dk$, this is a.
- The result is of type `Type::Real`.

Method `abs`: absolute value of a quaternion

`abs(`*`dom`* `x)`

- This method overloads `abs`.
- Returns the absolute value of $x := a + bi + cj + dk$, this is $\sqrt{a^2 + b^2 + c^2 + d^2}$.
- The result is of type `Type::Real`.

Method `conjugate`: conjugate element

`conjugate(`*`dom`* `x)`

- This method overloads `conjugate`.
- Returns the conjugate of $x := a + bI + cJ + dK$, which is defined to be $a - bI - cJ - dK$.

Method `intpower`: multiplies quaternions

`intpower(`*`dom`* `x, DOM_INT)`

- This method computes x^n for integers `n`.
- The implementation uses "repeated squaring".
- `Dom::Quaternion` is used by `"_power"`.

Method `nthroot`: n-th root of a quaternion

```
nthroot(dom x, DOM_INT n)
```

- This method computes the `n`-th root of `x`.
- The implementation uses "repeated squaring".
- `Dom::Quaternion` is used by `"_power"`.

Method `norm`: norm of a quaternion

```
norm(dom x)
```

- This method overloads `norm`.
- Returns the norm of $x := a + bi + cj + dk$, this is $a^2 + b^2 + c^2 + d^2$.
- The result is of type `Type::Real`.

Method `random`: random number generation

```
random()
```

- This methods returns a randomly generated quaternion $x := a + bi + cj + dk$ with `a, b, c` and `d` being nonnegative numbers with at most 12 digits generated by `random`.

Method `scalarmult`: scalar multiplication

```
scalarmult(Type::Real s, dom x)
```

- Returns the product sx.

Method `scalarprod`: inner product

```
scalarprod(dom x, dom y)
```

- Returns the inner product of $x := a + bi + cj + dk$ and $y := e + fi + gj + hk$, this is: $ae + bf + cg + dh$.

Method `sign`: sign of a quaternion

```
sign(dom x)
```

- This method overloads `sign`.
- Returns the sign of $x := a + bi + cj + dk$, this is $\frac{1}{abs(x)}x$.
- The result is of type `Type::Real`.

Conversion Methods

Method `convert`: conversion of objects

```
convert(any x)
```

- This method tries to convert `x` into an element of `Dom::Quaternion`. If `x` is a list, it must consist of four real numbers (type `Type::Real`). Constant real expressions and the identifiers`i`, `J`, `j`, `K`, and `k` can also be converted to domain elements, as well as sums and products of them. A matrix of the type `Dom::Matrix(Dom::Complex)` can be converted if and only if it is of the special form:

$$\begin{pmatrix} a+bI & -c-dI \\ c-dI & a-bI \end{pmatrix}$$

  If the conversion fails, `FAIL` is returned.

Method `convert_to`: conversion to other domains

```
convert_to(dom x, any T)
```

- This method tries to convert the number `x` to an element of type `T`, or, if `T` is not a domain, to the domain type of `T`. If the conversion fails, `FAIL` is returned.
- It currently handles the following domains for `T`: `DOM_EXPR`, `DOM_LIST`, `Dom::Matrix(Dom::Complex)`.

Method `expr`: converts a quaternion to an object of a kernel domain

```
expr(dom x)
```

- This method overloads `expr`.
- This method converts `x` into an expression of the form `a+bI+cJ+dK`.
- The result is an object of the kernel domain `DOM_EXPR`.
- This method overloads the function `expr` for quaternions, i.e., you may use it in the form `expr(x)`.

Method `matrixform`: converts a quaternion to a 2 x 2 matrix with complex entries.

```
matrixform(dom x)
```

- This method converts $x := a + bI + cJ + dK$ to the 2×2-matrix:

$$\begin{pmatrix} a+bI & -c-dI \\ c-dI & a-bI \end{pmatrix},$$

- The result is an object of the domain `Dom::Matrix(Dom::Complex)`.

Technical Methods

Method `TeX`: generate TeX-formatted string

`TeX(`*`dom`* `x)`

- This method overloads `generate::TeX`.
- `Dom::Quaternion(x)` returns a TeX-formatted string representing `x`.

Method `map`: apply a function to all components of a quaternion

`map(`*`dom`* `x,` *`function`* `f,` *`any`* `arg, ...)`

- This method overloads `map`.
- `Dom::Quaternion(x, f)` returns a copy of `x` where each component `co` of `x` has been replaced by `f(co)`. So for the quaternion $x := a + bi + cj + dk$, `Dom::Quaternion(x, f)` returns the quaternion $f(a) + f(b)i + f(c)j + f(d)k$.
- If optional arguments are present, then each component `co` of `x` is replaced by f(co, arg...). So for the quaternion $x := a + bi + cj + dk$, `Dom::Quaternion(x, f, arg, ...)` returns the quaternion $f(a, arg, \ldots) + f(b, arg, \ldots)i + f(c, arg, \ldots)j + f(d, arg, \ldots)k$.

Method `simplify`: simplification of a quaternion

`simplify(`*`dom`* `x)`

- This method overloads `simplify`.
- Tries to simplify the quaternion $x := a + bi + cj + dk$ by trying to simplify every component `a, b, c, d` of `x`.

Example 1. Creating some quaternions.

```
>> Dom::Quaternion([1,2,3,4]),
   Dom::Quaternion(11+12*i+13*j+14*k);
   M := Dom::Matrix(Dom::Complex)([[3+4*I,-6-2*I],[6-2*I,3-
4*I]]):
   M, Dom::Quaternion(M)
```

```
      3 J + 4 K + (1 + 2 I), 13 J + 14 K + (11 + 12 I)

       +-                    -+
       |  3 + 4 I, - 6 - 2 I  |
       |                      |, 6 J + 2 K + (3 + 4 I)
       |  6 - 2 I,  3 - 4 I   |
       +-                    -+
```

Example 2. Doing some standard arithmetic.

```
>> a:=Dom::Quaternion([1,2,3,4]):
   b:=Dom::Quaternion([11,2,33.3,2/3]):
   a*b, a+b, a^2/3, b^3;
```

```
 72.96666667 J + 105.2666667 K - (95.56666667 + 107.2 I),

              14 K                        8 K
    36.3 J + ---- + (12 + 4 I), 2 J + --- - (28/3 - 4/3 I),
              3                          3

    - 24986.137 J - 500.222963 K - (35409.03666 + 1500.668889 I)
```

Example 3. More mathematical operations:

```
>> a:=Dom::Quaternion([1,2,3,4]):
   b:=Dom::Quaternion([11,2,33.3,2/3]):
   Dom::Quaternion::nthroot(b,3);
   abs(a), sign(b)
```

```
 1.325212827 J + 0.02653078732 K +

    (2.993953193 + 0.07959236197 I)

   1/2
 30   , 0.9478242358 J + 0.01897546018 K +

    (0.3130950929 + 0.05692638053 I)
```

Example 4. Some miscellaneous operations.

```
>> a:=Dom::Quaternion([1,2,3,4]):
   Dom::Quaternion::matrixform(a);
   map(a, sqrt), map(a, _plus, 1);
```

```
                   +-                     -+
                   |  1 + 2 I, - 3 - 4 I   |
                   |                       |
                   |  3 - 4 I,  1 - 2 I    |
                   +-                     -+

                 1/2      1/2
       2 K + I 2    + J 3    + 1, 4 J + 5 K + (2 + 3 I)
```

Super-Domain: `Dom::BaseDomain`

Axioms

`Ax::canonicalRep`

Changes:

- No changes.

`Dom::Rational` – the field of rational numbers

`Dom::Rational` is the field of rational numbers represented by elements of the domains `DOM_INT` or `DOM_RAT`.

Creating Elements:

- `Dom::Rational(x)`

Parameters:

`x` — an integer or a rational number

Categories:

`Cat::QuotientField(Dom::Integer)`,
`Cat::DifferentialRing`, `Cat::OrderedSet`

Related Domains: `Dom::Complex`, `Dom::Float`, `Dom::Numerical`, `Dom::Rational`, `Dom::Real`

Details:

- `Dom::Rational` is the domain of rational numbers represented by expressions of type `DOM_INT` or `DOM_RAT`.
- Elements of `Dom::Rational` are usually not created explicitly. However, if one creates elements using the usual syntax, it is checked wether the input is of type `DOM_INT` or `DOM_RAT`. This means `Dom::Rational` is a façade domain which creates elements of domain type `DOM_INT` or `DOM_RAT`.
- Viewed as a differential ring `Dom::Rational` is trivial, it contains constants only.

- `Dom::Rational` has the domain `Dom::Numerical` as its super domain, i.e., it inherits each method which is defined by `Dom::Numerical` and not re-implemented by `Dom::Rational`. Methods described below are re-implemented by `Dom::Rational`.

Mathematical Methods

Method `denom`: denominator of a rational number

`denom(`*`dom`* `x)`

- This method returns the denominator of the rational number `x`.
- This method overloads `denom`.

Method `diff`: differentiates

`diff(`*`dom`* `z, <,` *`any`* `x, ...> )`

- This method returns `z` if it is called with only one argument. Otherwise it returns 0.
- This method overloads `diff`.

Method `numer`: numerator of the rational number

`numer(`*`dom`* `x)`

- This method returns the numerator of the rational number `x`.
- This method overloads `numer`.

Method `random`: random number generation

`random()`

- This methods returns a rational number `a/b` where `a` and `b` are integers between -999 and 999.

Method `retract`: retract to an integer element

`retract(`*`dom`* `x)`

- This method returns `x` if it is an integer and `FAIL` otherwise.

Conversion Methods

Method `convert`: conversion of objects

```
convert(any x)
```

- This method tries to convert `x` to a number of type `Dom::Rational`. This is only possible if `x` is of type `DOM_INT` or `DOM_RAT`. If the conversion fails, `FAIL` is returned.

Method `convert_to`: conversion to other domains

```
convert_to(dom x, any T)
```

- This method tries to convert the number `x` to an element of type `T`, or, if `T` is not a domain, to the domain type of `T`. If the conversion fails, `FAIL` is returned.
- The following domains are allowed for `T`: `DOM_INT`, `Dom::Integer`, `Dom::Rational`, `DOM_RAT`, `DOM_FLOAT`, `Dom::Float` and `Dom::Numerical`.

Method `testtype`: type checking

```
testtype(any x, dom T)
```

- This method checks whether it can convert `x` to the domain `Dom::Rational`. This is the case if `x` is of type `DOM_INT` or `DOM_RAT`. It returns `TRUE` if it can perform the conversion. Otherwise `FAIL` is returned.
- In general this method is called from the function `testtype` and not directly by the user. Example 2 demonstrates this behaviour.

Example 1. Creating some rational numbers using `Dom::Rational`. This example also shows that `Dom::Rational` is a façade domain.

```
>> Dom::Rational(2/3) ; domtype(%)
                                    2/3

                                 DOM_RAT

>> Dom::Rational(2.0)

 Error: illegal arguments [Dom::Rational::new]
```

Example 2. By tracing the method `Dom::Rational::testtype` we can see the interaction between `testtype` and `Dom::Rational::testtype`.

```
>> prog::trace(Dom::Rational::testtype):
   delete x:
   testtype(x, Dom::Rational);
   testtype(3/4, Dom::Rational);
   prog::untrace(Dom::Rational::testtype):

 enter 'Dom::Rational::testtype'        with args   : x, Dom::Rational
 leave 'Dom::Rational::testtype'        with result : FAIL

                                FALSE
 enter 'Dom::Rational::testtype'        with args   : 3/4, Dom::Rational
 leave 'Dom::Rational::testtype'        with result : TRUE

                                TRUE
```

Super-Domain: `Dom::Numerical`

Axioms

`Ax::canonicalRep`, `Ax::systemRep`, `Ax::canonicalOrder`,
`Ax::efficientOperation("_divide")`,
`Ax::efficientOperation("_mult")`,
`Ax::efficientOperation("_invert")`

Changes:

- No changes.

`Dom::Real` – the field of real numbers

`Dom::Real` is the field of real numbers represented by elements of the kernel domains `DOM_INT`, `DOM_RAT`, `DOM_FLOAT` and `DOM_EXPR`.

Creating Elements:

- `Dom::Real(x)`

Parameters:

`x` — An expression of type `DOM_INT`, `DOM_RAT` or `DOM_FLOAT`. An expression of type `DOM_EXPR` is also allowed if it is of type `Type::Arithmetical` and if it contains no indeterminates which are not of type `Type::ConstantIdents` and if it contains no imaginary part.

Categories:

`Cat::DifferentialRing`, `Cat::Field`, `Cat::OrderedSet`

Related Domains: `Dom::Complex`, `Dom::Float`, `Dom::Integer`, `Dom::Numerical`, `Dom::Rational`

Details:

- `Dom::Real` is the domain of real numbers represented by expressions of type `DOM_INT`, `DOM_RAT` or `DOM_FLOAT`. An expression of type `DOM_EXPR` is considered as a real number if it is of type `Type::Arithmetical` and if it contains no indeterminates which are not of type `Type::ConstantIdents` and if it contains no imaginary part, cf. Example 2.
- `Dom::Real` has category `Cat::Field` because of pragmatism. This domain actually is not a field because `bool(1.0 = float(3) / float(3))` returns `FALSE` for example.
- Elements may not have an unique representation, for example `bool(0 = sin(2)^2 + cos(2)^2 - 1)` returns `FALSE`.
- Elements of `Dom::Real` are usually not created explicitly. However, if one creates elements using the usual syntax, it is checked whether the input expression can be converted to a number. This means `Dom::Real` is a façade domain which creates elements of domain type `DOM_INT`, `DOM_RAT`, `DOM_FLOAT` or `DOM_EXPR`.
- `Dom::Real` has no normal representation, because `0` and `0.0` both represent zero.
- Viewed as a differential ring, `Dom::Real` is trivial, it contains constants only.
- `Dom::Real` has the domain `Dom::Numerical` as its super domain, i.e., it inherits each method which is defined by `Dom::Numerical` and not re-implemented by `Dom::Real`. Methods described below are re-implemented by `Dom::Real`.

Mathematical Methods

Method `_less`: Boolean operator "less"

`_less(`*`dom`* `x, `*`dom`* `y)`

- Behaves like the Boolean operator `_less`.

Method `_leequal`: Boolean operator "less or equal"

`_leequal(`*`dom`* `x, `*`dom`* `y)`

- Behaves like the Boolean operator `_leequal`.

Method `_power`: power operator

`_power(`*`dom`* `z, `*`any`* `n)`

- This method returns z^n if `n` is an integer, a floating point or a rational number or if `n` can be converted to a `Dom::Real`. Otherwise, an error message is issued.

Method `conjugate`: complex conjugate

`conjugate(`*`dom`* `x)`

- This method returns `x`.

Method `Im`: imaginary part of a real number

`Im(`*`dom`* `x)`

- This method returns 0.

Method `random`: random number generation

`random()`

- This methods returns a randomly generated positive integer between 0 and $10^{12} - 1$.

`random(`*`integer`* `n)`

- This method returns a random number generator which creates positive integer between 0 and $n - 1$.

`random(`*`integer`* `m..`*`integer`* `n)`

- This method returns a random number generator which creates positive integer between m and n.

Method `Re`: real part of a real number

`Re(`*`dom`* `x)`

- This method returns `x`.

Conversion Methods

Method `convert`: conversion of objects

`convert(`*`any`* `x)`

- This method tries to convert `x` to a number of type `Dom::Real`. If the conversion fails, then `FAIL` is returned.

Method `convert_to`: conversion to other domains

`convert_to(`*`dom`* `x,` *`any`* `T)`

- This method tries to convert the number `x` to an element of type `T`, or, if `T` is not a domain, to the domain type of `T`. If the conversion fails, then `FAIL` is returned.
- The following domains are allowed for `T`: `DOM_INT`, `Dom::Integer`, `DOM_RAT`, `Dom::Rational`, `DOM_FLOAT`, `Dom::Float`, `Dom::Numerical`, `Dom::ArithmeticalExpression`, `Dom::Complex`.

Example 1.

```
>> Dom::Real(2/3)
                                  2/3
>> Dom::Real(0.5666)
                               0.5666
```

Example 2.

```
>> Dom::Real(PI)
                                  PI
>> Dom::Real(sin(2))
                               sin(2)
>> Dom::Real(sin(2/3 * I) + 3)
 Error: illegal arguments [Dom::Real::new]
>> Dom::Real(sin(x))
 Error: illegal arguments [Dom::Real::new]
```

Super-Domain: `Dom::Complex`

Axioms

```
Ax::systemRep, Ax::canonicalOrder,
Ax::efficientOperation("_divide"),
Ax::efficientOperation("_mult"),
Ax::efficientOperation("_invert")
```

Changes:

- No changes.

`Dom::SparseMatrixF2` – the domain of sparse matrices over the field with two elements

`Dom::SparseMatrixF2` represents the set of all matrices over the finite field with two elements.

Domain:

- `Dom::SparseMatrixF2`

Details:

- `Dom::SparseMatrixF2` is mathematically equivalent to `Dom::Matrix(Dom::IntegerMod(2` However, the internal representation guarantees that both storage and computing time required for the arithmetical operations depend on the number of nonzero entries. Therefore, `Dom::SparseMatrixF2` should be used for matrices with few nonzero entries.

`Dom::SparseMatrixF2(m, n, [s1, ..., sm])` creates the m times n matrix (a_{ij}) such that, for each i, the set of all j with $a_{ij} = 1$ equals the set (or list) s_i.

Creating Elements:

- `Dom::SparseMatrixF2(m, n, [s1, ...])`
- `Dom::SparseMatrixF2(m, n, f)`

Parameters:

`m, n` — positive integers
`s1, ...` — sets or lists of integers between 1 and `n`
`f` — a procedure or another object that, when called with an integer between 1 and *m* and another integer between 1 and *n*, returns an element of `Dom::IntegerMod(2)`.

Categories:

`Cat::Matrix(Dom::IntegerMod(2))`

Related Domains: `Dom::Matrix`

Entries:

`coeffRing` The coefficient ring always equals `Dom::IntegerMod(2)`.

`isSparse` This entry is always set to `TRUE`.

Mathematical Methods

Method `zeroMatrix`: matrix of a given dimension, consisting of zeros

`zeroMatrix(`*`integer`* `m, `*`integer`* `n)`

- This method returns the *m* times *n* matrix whose entries are all zero.

Method `_plus`: add matrices

`_plus(`*`dom`* `A, ...)`

- This method returns the sum of the given matrices. All matrices must have the same dimensions.
- This method overloads the function `_plus`.

Method `_negate`: negate a matrix

`_negate(`*`dom`* `A)`

- Because the coefficient field has characteristic two, the negative of `A` is just `A` itself.
- This method overloads `_negate`.

Method `matrixvectorproduct`: multiply a matrix and a vector

`matrixvectorproduct(`*`dom`* `A, `*`dom`* `b)`

- This method returns the product of the matrix `A` and the column vector `b`; the number of columns of `A` must be the same as the number of rows of `b`.

Method `_mult`: multiply a matrix and a vector

`_mult(`*`dom`* `A, `*`dom`* `b)`

- This method does the same as `"matrixvectorproduct"`.
- It overloads the function `_mult`.
- The product of arbitrary sparse matrices (where `b` is not a vector) has *not* been implemented.

Method `randmatrix`: generate random matrix

`randmatrix(`*`integer`* `m, `*`integer`* `n <`*`integer`* `s>)`

- This method generates an m times n random matrix by defining in each row s entries to be equal to 1; these are drawn with repetition such that the actual number of ones may be less. If `s` is not given, it defaults to 6.

Access Methods

Method `nrows`: number of rows

`nrows(`*`dom`* `A)`

- This method returns the number of rows of `A`.

Method `ncols`: number of rows

`ncols(`*`dom`* `A)`

- This method returns the number of columns of `A`.

Method `dimen`: number of rows and columns

`dimen(`*`dom`* `A)`

- This method returns a list of two elements, the first being the number of rows, the second being the number of columns of `A`.

Method `body`: body of the matrix

```
body(dom A)
```

- This method returns a list of sets, where the *i*-th set is the set of the index positions of all ones in the *i*-th row.

Method `row`: row of a matrix

```
row(dom A, integer i)
```

- This method returns the *i*-th row of the *m* times *n* matrix `A` as a 1 times *n* matrix of type `Dom::SparseMatrixF2`.

Method `col`: column of a matrix

```
col(dom A, integer i)
```

- This method returns the *i*-th column of the *m* times *n* matrix `A` *as a row vector*, i.e. as a 1 times *m* matrix of type `Dom::SparseMatrixF2`.

Method `_index`: row or single entry of a matrix

```
_index(dom A, integer i)
```

- returns the *i*-th row of `A`.
- This method overloads the `_index` operator; `A[i]` may be entered equivalently.

```
_index(dom A, integer i, integer j)
```

- returns the entry of `A` in the *i*-th row, *j*-th column.
- Equivalently, `A[i,j]` may be entered.

Method `set_index`: assignment to a matrix entry

```
set_index(dom A, integer i, integer j, value v)
```

- This method returns a copy of `A`, with the result of converting `v` into an element of `Dom::IntegerMod(2)` entered in the *i*-th row, *j*-th column. `Dom::IntegerMod(2)` must be able to convert `v` into a field element.
- This method can be used for indexed assignments using the syntax `A[i,j]:=v`. In this case, the value of the identifier or local variable `A` is changed as a side effect; `v` (but not the result of converting it to a field element!) is returned.
  If the assignment stops with an error, the domain element stored in `A` is destroyed, and the new value of `A` is `FAIL`.

- See `_assign` for more information about indexed assignments.

Conversion Methods

Method `convert_to`: conversion of a sparse matrix into another type

```
convert_to(dom A, any T)
```

- This method tries to convert `A` into type `T`. Currently only a conversion into a `Dom::Matrix(Dom::IntegerMod(2))` is possible.

Example 1. We create a sparse matrix with three nonzero entries:

```
>> A:=Dom::SparseMatrixF2(3, 3, [{2}, {1}, {3}])
```

```
                          [{2}, {1}, {3}]
```

Conversion to a `Dom::Matrix` yields a nicer output, but now nine entries have to be stored:

```
>> A::dom::convert_to(A, Dom::Matrix(Dom::IntegerMod(2)))
```

```
                 +-                            -+
                 |  0 mod 2, 1 mod 2, 0 mod 2  |
                 |                             |
                 |  1 mod 2, 0 mod 2, 0 mod 2  |
                 |                             |
                 |  0 mod 2, 0 mod 2, 1 mod 2  |
                 +-                            -+
```

Super-Domain: `Dom::BaseDomain`

Axioms

```
Ax::canonicalRep
```

Changes:

- `Dom::SparseMatrixF2` is a new domain.

`Dom::SquareMatrix` – the rings of square matrices

`Dom::SquareMatrix(n, R)` creates the ring of $n \times n$ matrices over the component ring R.

Domain:

- `Dom::SquareMatrix(n<, R>)`

Parameters:

`n` — a positive integer
`R` — a ring, i.e., a domain of category `Cat::Rng`; default is `Dom::ExpressionField()`

Details:

- `Dom::SquareMatrix(n, R)` creates a domain which represents the ring of $n \times n$ matrices over a component domain `R`. The domain `R` must be of category `Cat::Rng` (a ring, possibly without unit).
- If the optional parameter `R` is not given, the domain `Dom::ExpressionField()` is used as the component ring for the square matrices.
- For matrices of a domain created by `Dom::SquareMatrix(n, R)`, standard matrix arithmetic is implemented by overloading the standard arithmetical operators `+, -, *, /` and `^`. All functions of the `linalg` package dealing with matrices can also be applied.
- `Dom::SquareMatrix(n, R)` has the domain `Dom::Matrix(R)` as its super domain, i.e., it inherits each method which is defined by `Dom::Matrix(R)` and not re-implemented by `Dom::SquareMatrix(n, R)`.

  Methods described below are re-implemented by `Dom::SquareMatrix`.
- The domain `Dom::Matrix(R)` represents matrices over `R` of arbitrary size, and it therefore does not have any algebraic structure (except of being a *set* of matrices).

  The domain `Dom::MatrixGroup(m, n, R)` represents the Abelian group of $m \times n$ matrices over R.

Creating Elements:

- `Dom::SquareMatrix(n, R)(Array)`
- `Dom::SquareMatrix(n, R)(Matrix)`
- `Dom::SquareMatrix(n, R)(<n, n>)`
- `Dom::SquareMatrix(n, R)(<n, n, >ListOfRows)`
- `Dom::SquareMatrix(n, R)(<n, n, >f)`
- `Dom::SquareMatrix(n, R)(<n, n, >List, Diagonal)`
- `Dom::SquareMatrix(n, R)(<n, n, >g, Diagonal)`
- `Dom::SquareMatrix(n, R)(<n, n, >List, Banded)`

Parameters:

`Array`	— an `n`×`n` array
`Matrix`	— an `n`×`n` matrix, i.e., an element of a domain of category `Cat::Matrix`
`List`	— a list of matrix components
`ListOfRows`	— a list of at most `n` rows; each row is a list of at most `n` matrix components
`f`	— a function or a functional expression with two parameters (the row and column index)
`g`	— a function or a functional expression with one parameter (the row index)

Options:

`Diagonal`	— create a diagonal matrix
`Banded`	— create a banded Toeplitz matrix

Categories:

`Cat::SquareMatrix(R)`

Related Domains: `Dom::Matrix`, `Dom::MatrixGroup`

Details:

- `Dom::SquareMatrix(n, R)(Array)` and `Dom::SquareMatrix(n, R)(Matrix)` create a new matrix formed by the entries of `Array` and `Matrix`, respectively.

  The components of `Array` and `Matrix`, respectively, are converted into elements of the domain `R`. An error message is issued if one of these conversions fails.

- The call `Dom::SquareMatrix(n, R)(<n, n>)` returns the $n \times n$ zero matrix. Note that the $n \times n$ zero matrix is also defined by the entry `"zero"` (see below).

- `Dom::SquareMatrix(n, R)(<n, n, >ListOfRows)` creates an $n \times n$ matrix with components taken from the nested list `ListOfRows`. Each inner list corresponds to a row of the matrix.

  If an inner list has less than `n` entries, the remaining components in the corresponding row of the matrix are set to zero. If there are less than `n` inner lists, the remaining lower rows of the matrix are filled with zeroes.

  The entries of the inner lists are converted into elements of the domain `R`. An error message is issued if one of these conversions fails.

- `Dom::SquareMatrix(n, R)(<n, n, >f)` returns the matrix whose (i, j)th component is the value of the function call `f(i, j)`. The row and column indices i and j range from 1 to `n`.

The function values are converted into elements of the domain R. An error message is issued if one of these conversions fails.

Option <*Diagonal*>:

- With the option *Diagonal*, diagonal matrices can be created with diagonal elements taken from a list, or computed by a function.
- Dom::SquareMatrix(n, R)(<n, n, >List, *Diagonal*) creates the $n \times n$ diagonal matrix whose diagonal elements are the entries of List.

  List must have at most n entries. If it has fewer elements, the remaining diagonal elements are set to zero.

  The entries of List are converted into elements of the domain R. An error message is issued if one of these conversions fails.
- Dom::SquareMatrix(n, R)(<n, n, >g, *Diagonal*) returns the matrix whose ith diagonal element is g(i), where the index i runs from 1 to n.

  The function values are converted into elements of the domain R. An error message is issued if one of these conversions fails.

Option <*Banded*>:

- Dom::SquareMatrix(n, R)(<n, n, >List, *Banded*) creates an $n \times n$ banded Toeplitz matrix with the elements of List as entries. The number of entries of List must be odd, say $2h+1$, and must not exceed n. The resulting matrix has bandwidth at most $2h+1$.

  All elements of the main diagonal of the created matrix are initialized with the middle element of List. All elements of the ith subdiagonal are initialized with the $(h+1-i)$th element of List. All elements of the ith superdiagonal are initialized with the $(h+1+i)$th element of List. All entries on the remaining sub- and superdiagonals are set to zero.

  The entries of List are converted into elements of the domain R. An error message is issued if one of these conversions fails.

Entries:

one	is the $n \times n$ identity matrix. This entry exists if the component ring R is a domain of category Cat::Ring, i.e., a ring with unit.
randomDimen	is set to [n, n].
zero	is the $n \times n$ zero matrix.

Mathematical Methods

Method `evalp`: evaluating matrices of polynomials at a certain point

```
evalp(dom A, equation x = a, ...)
```

- This method evaluates the polynomial components of `A` at the point `x = a`. See the system function `evalp` for details.

  The matrix returned is of the domain `Dom::SquareMatrix(n, R::coeffRing)`, if the evaluation of each component leads to an element of the coefficient ring of the polynomial domain. Otherwise the matrix returned is of the domain of `A`.
- This method is only defined if `R` is a polynomial ring of category `Cat::Polynomial`.
- This method overloads the function `evalp` for matrices, i.e., one may use it in the form `evalp(A, x = a)`.

Method `identity`: identity matrix

```
identity(positive integer k)
```

- This method returns the $k \times k$ identity matrix.
- The matrix returned is of the domain `Dom::Matrix(R)` if `k`$\neq$`n`.

- This method only exists if the component ring `R` is of category `Cat::Ring`, i.e., a ring with unit.

Method `matdim`: matrix dimension

```
matdim(dom A)
```

- This method returns the list `[n, n]`, i.e., the matrix dimension of `A`.

Method `random`: random matrix generation

```
random()
```

- This method returns an $n \times n$ random matrix.
- The components of the random matrix are generated with the method `"random"` of the component ring `R`.

Access Methods

Method `_concat`: horizontal concatenation of matrices

`_concat(`*dom* `A, `*dom* `B, ...)`

- This method appends the matrices `B, ...` to the right side of the matrix `A`.
- An error message is issued if the given matrices do not have the same number of rows.
- The matrix returned is of the domain `Dom::Matrix(R)`!

- This method overloads the function `_concat` for matrices, i.e., one may use it in the form `A . B . ...`, or in functional notation: `_concat(A, B, ...)`.

Method `_index`: matrix indexing

`_index(`*dom* `A, `*row index* `i, `*column index* `j)`

- Returns the (i, j)th entry of the matrix `A`.

`_index(`*dom* `A, `*row-range* `r1..r2, `*column-range* `c1..c2)`

- Returns the submatrix of `A`, created by the rows of `A` with indices from `r1` to `r2` and the columns of `A` with indices from `c1` to `c2`.
- The submatrix returned is of the domain `Dom::Matrix(R)`!

- This method overloads the function `_index` for matrices, i.e., one may use it in the form `A[i, j]` and `A[r1..r2, c1..c2]`, respectively or in functional notation: `_index(A, ...)`.

Method `concatMatrix`: horizontal concatenation of matrices

`concatMatrix(`*dom* `A, `*dom* `B, ...)`

- This method is identical to the method `"_concat"`.

Method `col`: extracting a column

`col(`*dom* `A, `*column index* `c)`

- This method extracts the column with index `c` of the matrix `A` and returns it as a column vector, i.e., as an element of the domain type `Dom::Matrix(R)`.
- An error message is issued if `c` is less than one or greater than `n`.

Method `delCol`: deleting a column

`delCol(`*`dom`* `A, `*`column index`* `c)`

- This method returns the matrix obtained by deleting the column with index `c` of the matrix `A`.
- `NIL` is returned if `A` only consists of one column.
- The matrix returned is of the domain `Dom::Matrix(R)`.

- An error message is issued if `c` is less than one or greater than `n`.

Method `delRow`: deleting a row

`delRow(`*`dom`* `A, `*`row index`* `r)`

- This method returns the matrix obtained by deleting the row with index `r` of the matrix `A`.
- `NIL` is returned if `A` only consists of one row.
- The matrix returned is of the domain `Dom::Matrix(R)`.

- An error message is issued if `r` is less than one or greater than `n`.

Method `row`: extracting a row

`row(`*`dom`* `A, `*`row index`* `r)`

- This method extracts the row with index `r` of the matrix `A` and returns it as a row vector, i.e., as an element of the domain `Dom::Matrix(R)`.
- An error message is issued if `r` is less than one or greater than `n`.

Method `stackMatrix`: vertical concatenation of matrices

`stackMatrix(`*`dom`* `A, `*`dom`* `B, ...)`

- This method stacks the matrix `A` on top of the matrix `B`. If further arguments are given, the result is stacked on the top of the third matrix, and so on.
- An error message is issued if the given matrices do not have the same number of columns.
- The matrix returned is of the domain `Dom::Matrix(R)`!

Conversion Methods

Method `create`: defining matrices without component conversions

`create(`*`any`* `x, ...)`

- This method creates a new matrix assuming that the components are of domain type `R`.
  See "Creating Elements" above for a complete description of the parameters, with one exception: one *cannot* use this method to create a matrix from a function or a functional expression.
- This method should be used if the elements of the parameters `x, ...` are elements of the domain type `R`. This is often the case if a matrix is to be created whose components come from preceding matrix and scalar operations.

Example 1. A lot of examples can be found on the help page of the domain constructor `Dom::Matrix`, and most of them are also examples for working with domains created by `Dom::SquareMatrix`.

These examples only concentrate on some differences with respect to working with matrices of the domain `Dom::Matrix(R)`.

The following command defines the ring of two-dimensional matrices over the rationals:

```
>> SqMatQ := Dom::SquareMatrix(2, Dom::Rational)

               Dom::SquareMatrix(2, Dom::Rational)

>> SqMatQ::hasProp(Cat::Ring)

                                 TRUE
```

The unit is defined by the entry `"one"`, which is the 2×2 identity matrix:

```
>> SqMatQ::one

                              +-      -+
                              |  1, 0  |
                              |        |
                              |  0, 1  |
                              +-      -+
```

Note that some operations defined by the domain `SqMatQ` return matrices which are no longer square. They return therefore matrices of the domain `Dom::Matrix(Dom::Rational)`, the super-domain of `SqMatQ`. For example, if we delete the first row of the matrix:

```
>> A := SqMatQ([[1, 2], [-5, 3]])
```

```
+-       -+
|   1, 2  |
|         |
|  -5, 3  |
+-       -+
```

we get the matrix:

```
>> SqMatQ::delRow(A, 1)
```

```
 +-     -+
 | -5, 3 |
 +-     -+
```

which is of the domain type:

```
>> domtype(%)
```

```
                  Dom::Matrix(Dom::Rational)
```

Example 2. We can convert a square matrix into or from another matrix domain, as shown next:

```
>> SqMatR := Dom::SquareMatrix(3, Dom::Real):
   MatC := Dom::Matrix(Dom::Complex):

>> A := SqMatR((i, j) -> sin(i*j))
```

```
                +-                         -+
                |  sin(1), sin(2), sin(3)  |
                |                          |
                |  sin(2), sin(4), sin(6)  |
                |                          |
                |  sin(3), sin(6), sin(9)  |
                +-                         -+
```

To convert `A` into a matrix of the domain `MatC`, enter:

```
>> coerce(A, MatC)
```

```
                +-                         -+
                |  sin(1), sin(2), sin(3)  |
                |                          |
                |  sin(2), sin(4), sin(6)  |
                |                          |
                |  sin(3), sin(6), sin(9)  |
                +-                         -+
```

```
>> domtype(%)
```

```
Dom::Matrix(Dom::Complex)
```

The conversion is done component-wise, as the following examples shows:

```
>> B := MatC([[0, 1], [exp(I), 0]])
```

```
+-           -+
|     0,   1  |
|             |
|  exp(I), 0  |
+-           -+
```

The matrix B is square but has one complex component and therefore cannot be converted into the domain `SqMatR`:

```
>> coerce(B, SqMatR)
```

```
FAIL
```

Super-Domain: `Dom::Matrix`

Axioms

if `R` has `Ax::canonicalRep`, then
`Ax::canonicalRep`

Changes:

- The method `"dimen"` was renamed to `"matdim"`.
- The method `"newThis"` was renamed to `"create"`.
- Some new methods were implemented or extended for the domain `Dom::Matrix`. See the corresponding help page for details (note that `Dom::SquareMatrix(n, R)` inherits every method which is defined for `Dom::Matrix(R)` and not re-implemented by `Dom::SquareMatrix(n, R)`.

`Dom::UnivariatePolynomial` – the domains of univariate polynomials

`Dom::UnivariatePolynomial(Var, R, ..)` creates the domain of univariate polynomials in the variable `Var` over the commutative ring `R`.

Domain:

- `Dom::UnivariatePolynomial(<Var <, R <, Order>>>)`

Parameters:

`Var` — an indeterminate given by an identifier; default is `x`.

`R` — a commutative ring, i.e. a domain of category `Cat::CommutativeRing`; default is `Dom::ExpressionField(normal)`.

`Order` — a monomial ordering, i.e. one of the predefined orderings `LexOrder`, `DegreeOrder` or `DegInvLexOrder` or an element of domain `Dom::MonomOrdering`; default is `LexOrder`.

Details:

- `Dom::UnivariatePolynomial` represents univariate polynomials over arbitrary commutative rings.

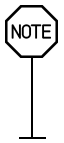
  All usual algebraic and arithmetical polynomial operations are implemented, including Gröbner basis computations.

- `Dom::UnivariatePolynomial(Var, R, Order)` creates a domain of univariate polynomials in the variable `Var` over a domain of category `Cat::CommutativeRing` in sparse representation with respect to the monomial ordering `Order`.

- `Dom::UnivariatePolynomial()` creates the univariate polynomial domain in the variable `x` over the domain `Dom::ExpressionField(normal)` with respect to the lexicographic monomial ordering.

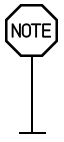
- `Dom::UnivariatePolynomial(Var)` creates the univariate polynomial domain in the variable `Var` over the domain `Dom::ExpressionField(normal)` with respect to the lexicographic monomial ordering.

- Only commutative coefficient rings of type `DOM_DOMAIN` which inherit from `Dom::BaseDomain` are allowed. If `R` is of type `DOM_DOMAIN` but inherits not from `Dom::BaseDomain`, the domain `Dom::ExpressionField(normal)` will be used instead.



- For this domain only identifiers are valid variables.

- It is highly recommend to use only coefficient rings with unique zero representation. Otherwise it may happen that, e.g., a polynomial division will not terminate or a wrong degree will be returned.



- Please note that for reasons of efficiency not all methods check their arguments, not even at the interactive level. In particular this is true for many access methods, converting methods and technical methods. Therefore,

using these methods inappropriately may result in strange error messages.

Creating Elements:

- `Dom::UnivariatePolynomial(Var, R, Order)(p)`
- `Dom::UnivariatePolynomial(Var, R, Order)(lm)`

Parameters:

`p` — a polynomial or a polynomial expression.
`lm` — list of monomials, which are represented as lists containing the coefficients together with the exponents or exponent vectors.

Categories:

`Cat::UnivariatePolynomial(R)`

Related Domains: `Dom::Polynomial`, `Dom::DistributedPolynomial`, `Dom::MultivariatePolynomial`

Entries:

`characteristic` The characteristic of this domain.

`coeffRing` The coefficient ring of this domain as defined by the parameter `R`.

`key` The name of the domain created.

`one` The neutral element w.r.t. `"_mult"`.

`ordering` The monomial order as defined by the parameter `Order`.

`variables` The list of the variable as defined by the parameter `Var`.

`zero` The neutral element w.r.t. `"_plus"`.

Access Methods

Method `coeff`: coefficients of a polynomial

`coeff(`*`dom`* `a)`
`coeff(`*`dom`* `a, Var,` *`NonNegativeInteger`* `n)`
`coeff(`*`dom`* `a,` *`NonNegativeInteger`* `n)`

- `coeff(a)` returns a sequence with all coefficients of `a` as elements of `R`. The coefficients are ordered according to the monomial ordering `Order`.

- `coeff(a, Var, n)` returns the coefficient of the term `Var^n` as an element of `R`.
- `coeff(a, n)` returns the coefficient of the term `Var^n` as an element of `R`.
- This method overloads the function `coeff` for polynomials.

Method `vectorize`: vectorized form of a polynomial

`vectorize(`*`dom`* `a <,` *`PositiveInteger`* `n>)`

- Returns `a` in its vectorized form, i.e. a list of *all* (zero and nonzero) coefficients of `a` as elements of `R` in increasing order. If `n` is explicitly given, whereby `n` must be greater than `degree(a)`, a list of `n` coefficient entries maybe filled up with zeros is returned.

Example 1. To create the ring of univariate polynomials in `x` over the integers one may define

```
>> UP:=Dom::UnivariatePolynomial(x,Dom::Integer)

      Dom::UnivariatePolynomial(x, Dom::Integer, LexOrder)
```

Now, let us create two univariate polynomials.

```
>> a:=UP((2*x-1)^2*(3*x+1))

                          3      2
                      12 x  - 8 x  - x + 1
```

```
>> b:=UP(((2*x-1)*(3*x+1))^2)

                     4       3       2
                 36 x  - 12 x  - 11 x  + 2 x + 1
```

The usual arithmetical operations for polynomials are available:

```
>> a^2+a*b

         7        6        5        4       3       2
    432 x  - 288 x  - 264 x  + 200 x  + 35 x  - 36 x  - x + 2
```

The leading coefficient, leading term, leading monomial and reductum of `a` are

```
>> lcoeff(a),lterm(a),lmonomial(a),UP::reductum(a)

                      3      3       2
                 12, x , 12 x , - 8 x  - x + 1
```

and `a` is of degree

```
>> degree(a)
```

```
                                3
```

The method `gcd` computes the greatest common divisor of two polynomials

```
>> gcd(a,b)
```

```
                           3      2
                       12 x  - 8 x  - x + 1
```

and `lcm` the least common multiple:

```
>> lcm(a,b)
```

```
                        4       3       2
                    36 x  - 12 x  - 11 x  + 2 x + 1
```

Computing the definite and indefinite integral of a polynomial is also possible,

```
>> int(a)
```

```
                         4        3        2
                      3 x  - 8/3 x  - 1/2 x  + x
```

which is in the case of indefinite integration simply the antiderivative of the polynomial.

```
>> D(int(a)), domtype(D(int(a)))
```

```
     3      2
 12 x  - 8 x  - x + 1, Dom::UnivariatePolynomial(x,

    Dom::Fraction(Dom::Integer), LexOrder)
```

But, since for representing the indefinite integral of `a` the coefficient ring chosen as the integers is not appropriate, the polynomial ring over its quotient field is used instead.

Furthermore, interpreting the polynomials as polynomial functions is also allowed in applying coefficient ring elements, polynomials of this domain or arbitrary expressions with option `Expr` to them:

```
>> a(5)
```

```
                               1296
```

```
>> a(b)
```

```
         12           11           10           9          8
 559872 x   - 559872 x   - 326592 x   + 414720 x  + 73872 x  -


          7         6          5         4         3
    123120 x  - 9924 x  + 18408 x  + 1144 x  - 1364 x  -

        2
    97 x  + 38 x + 4
```

```
>> a(sin(x),Expr)
```

```
                        3          2
              12 sin(x)  - 8 sin(x)  - sin(x) + 1
```

To get a vector of coefficients of a polynomial, which gives the dense representation of it, one may use the method `vectorize`.

```
>> UP::vectorize(a), UP::vectorize(a,6)
```

```
             [1, -1, -8, 12], [1, -1, -8, 12, 0, 0]
```

Super-Domain: `Dom::MultivariatePolynomial`

Axioms

if R has `Ax::normalRep`, then
`Ax::normalRep`

if R has `Ax::canonicalRep`, then
`Ax::canonicalRep`

Changes:

- `Dom::UnivariatePolynomial` is a new domain.