The Standard Library

Table of contents

Preface xii

Help Pages

`glossary` — glossary 1

`mathematical constants and functions` — an overview 3

`options` — options used by MuPAD functions 5

`:=` — assign variables 18

`.` — concatenate objects 22

`..` — range operator 25

`=, <>` — equations and inequalities 26

`<, <=, >, >=` — inequalities 29

`+` — add expressions 32

`-` — subtract expressions 39

`*` — multiply expressions 44

`/` — divide expressions 51

`^` — raise an expression to a power 55

`@` — compose functions 58

`@@` — iterate a function 60

`$` — create an expression sequence 61

`_exprseq` — expression sequences 65

`_index` — indexed access 67

`intersect, minus, union` — operators for sets and intervals 71

`_invert` — the reciprocal of an expression 74

`_lazy_and, _lazy_or` — “lazy evaluation” of Boolean expressions . . . 76

`_negate` — the negative of an expression 79

`_stmtseq` — statement sequences 82

`%if` — conditional creation of code by the parser 84

`Ci` — the cosine integral function 86

D — differential operator for functions 89

DIGITS — the significant digits of floating point numbers 95

Ei — the exponential integral function 101

FAIL — indicate a failed computation 104

FILEPATH — the pathname of a file that is currently loaded 105

HISTORY — the maximal number of elements in the history table 106

LEVEL — substitution depth of identifiers 107

Line-Editor — editing lines in the terminal version of MuPAD 112

MAXDEPTH — prevent infinite recursion during procedure calls 114

MAXLEVEL — prevent infinite recursion during evaluation 115

NIL — the singleton element of the domain DOM_NIL 117

NOTEBOOKFILE, NOTEBOOKPATH — Notebook file name and path 119

O — the domain of order terms (Landau symbols) 120

ORDER — the default number of terms in series expansions 123

path variables — file search paths 125

PRETTYPRINT — control the formatting of output 128

Re, Im — real and imaginary part of an arithmetical expression 130

RootOf — the set of roots of a polynomial 133

Si — the sine integral function 136

TESTPATH — write path for prog::test 138

TEXTWIDTH — the maximum number of characters in an output line . . 138

TRUE, FALSE, UNKNOWN — Boolean constants 140

UNIX — MuPAD command line options and initialization files for UNIX 141

abs — the absolute value of a real or complex number 145

alias, unalias — defines or un-defines an abbreviation or a macro . . 147

anames — identifiers that have values or properties 155

and, or, not, xor, ==>, <=> — Boolean operators 157

append — add elements to a list 162

arcsin, arccos, arctan, arccsc, arcsec, arccot — the inverse trigonometric functions 164

arcsinh, arccosh, arctanh, arccsch, arcsech, arccoth — the inverse hyperbolic functions 169

`arg` — the argument (polar angle) of a complex number 173
`args` — access procedure parameters 176
`array` — create an array 179
`assert` — assertions for debugging 185
`assign` — perform assignments given as equations 187
`assignElements` — assign values to entries of an array, a list, or a table 189
`assume` — attach a property to an identifier 192
`asympt` — compute an asymptotic series expansion 198
`bernoulli` — the Bernoulli numbers and polynomials 201
`besselI, besselJ, besselK, besselY` — the Bessel functions 204
`beta` — the beta function 209
`binomial` — binomial coefficients 211
`bool` — Boolean evaluation 214
`break` — terminate a loop or a `case` switch prematurely 218
`builtin` — representatives of C-functions of the MuPAD kernel 219
`bytes` — the memory used by the current MuPAD session 222
`card` — the cardinality of a set 223
`case` — switch statement 225
`ceil, floor, round, trunc` — rounding to an integer 228
`coeff` — the coefficients of a polynomial 231
`coerce` — type conversion 235
`collect` — collect coefficients of a polynomial expression 239
`combine` — combine terms of the same algebraic structure 242
`copyClosure` — copies the lexical closure of a procedure 247
`complexInfinity` — complex infinity 249
`conjugate` — complex conjugation 250
`contains` — test if an entry exists in a container 252
`content` — the content of a polynomial 256
`context` — evaluate an object in the enclosing context 259
`contfrac` — the domain of continued fractions 262
`debug` — execute a procedure in single-step mode 271

degree — the degree of a polynomial . . . 273

degreevec — the exponents of the leading term of a polynomial . . . 275

delete — delete the value of an identifier . . . 277

denom — the denominator of a rational expression . . . 280

diff — differentiate an expression or a polynomial . . . 281

dilog — the dilogarithm function . . . 287

dirac — the Dirac delta distribution . . . 290

discont — discontinuities of a function . . . 293

div — the integer part of a quotient . . . 297

divide — divide polynomials . . . 298

domtype — the data type of an object . . . 301

end — close a block statement . . . 304

erf, erfc — the error function and the complementary error function . 305

error — raise a user-specified exception . . . 308

eval — evaluate an object . . . 309

evalassign — assignment with evaluation of the left hand side . . . 316

evalp — evaluate a polynomial at a point . . . 318

exp — the exponential function . . . 320

expand — expand an expression . . . 323

export, unexport — export library functions or undo the export . . . 331

expose — display the source code of a procedure or the entries of a domain . . . 334

expr — convert into an element of a basic domain . . . 337

expr2text — convert objects into character strings . . . 341

external — create a module function environment . . . 344

extnops — the number of operands of a domain element . . . 346

extop — the operands of a domain element . . . 347

extsubsop — substitute operands of a domain element . . . 350

fact — the factorial function . . . 353

factor — factor a polynomial into irreducible polynomials . . . 355

fclose — close a file . . . 363

fileIO — an overview of MuPAD's file I/O functions . . . 364

finput — read MuPAD objects from a file 367
float — convert to a floating point number 370
fname — get a file name 375
fopen — open a file 376
for — for loop 380
fprint — write data to a file 385
frac — the fractional part of a number 390
frame — create a new frame, change to an existing frame 392
frandom — generate random floating point numbers 405
fread — read and execute a file 408
freeze, unfreeze — create an inactive or active copy of a function . . 411
ftextinput — read a text file 416
funcenv — create a function environment 420
gamma — the gamma function 424
gcd — the greatest common divisor of polynomials 427
gcdex — the extended Euclidean algorithm for polynomials 429
genident — create an unused identifier 430
genpoly — create a polynomial using the "b"-adic expansion 431
getpid — the process ID of the running MuPAD kernel 433
getprop — query properties of expressions 434
ground — ground term (constant coefficient) of a polynomial 437
has — check if an object occurs in another object 439
hastype — test if an object of a specified type occurs in another object 443
heaviside — the Heaviside step function 446
help — display a help page 449
history — access an entry of the history table 452
hold — delay evaluation 455
..., hull — convert to a floating point interval 461
hypergeom — the hypergeometric functions 463
icontent — the content of a polynomial with rational coefficients . . . 470
if — branch statement 472

id — the identity map 475
ifactor — factor an integer into primes 476
igamma — the incomplete Gamma function 482
igcd — the greatest common divisor of integers 485
igcdex — the extended Euclidean algorithm for two integers 487
ilcm — the least common multiple of integers 488
in — membership 489
indets — the indeterminates of an expression 493
indexval — indexed access to arrays and tables without evaluation . . 496
interpolate — polynomial interpolation 499
infinity — infinity 505
info — prints short information 506
input — interactive input of MuPAD objects 508
int — definite and indefinite integration 512
int2text — convert an integer to a character string 520
interval — convert constant subexpressions to intervals 521
irreducible — test irreducibility of a polynomial 523
is — check a mathematical property of an expression 524
isprime — primality test 529
isqrt — integer square root 531
iszero — generic zero test 532
ithprime — the i-th prime number 535
lambertV, lambertW — lower and upper real branch of the Lambert function 536
last — access a previously computed object 539
lasterror — reproduce the last error 543
lcm — the least common multiple of polynomials 545
lcoeff — the leading coefficient of a polynomial 546
ldegree — the lowest degree of the terms in a polynomial 549
length — the “length” of a MuPAD object (heuristic complexity) 550
level — evaluate an object with a specified substitution depth 553

`lhs, rhs` — the left, respectively right hand side of equations, inequalities, relations, intervals, and ranges . . . 560

`limit` — compute a limit . . . 561

`linsolve` — solve a system of linear equations . . . 565

`lllint` — compute an LLL-reduced basis of a lattice . . . 569

`lmonomial` — the leading monomial of a polynomial . . . 572

`ln` — the natural logarithm . . . 575

`loadlib` — load a library package . . . 578

`loadmod` — load a module . . . 580

`loadproc` — load an object on demand . . . 583

`log` — the logarithm to an arbitrary base . . . 585

`lterm` — the leading term of a polynomial . . . 588

`match` — pattern matching . . . 590

`matrix` — create a matrix or a vector . . . 597

`map` — apply a function to all operands of an object . . . 611

`mapcoeffs` — apply a function to the coefficients of a polynomial . . . 618

`maprat` — apply a function to the "rationalization" of an expression . . . 621

`max` — the maximum of numbers . . . 623

`min` — the minimum of numbers . . . 626

`mod, modp, mods` — the modulo functions . . . 628

`multcoeffs` — multiply the coefficients of a polynomial with a factor . . . 632

`new` — create a domain element . . . 634

`newDomain` — create a new data type (domain) . . . 637

`next` — skip a step in a loop . . . 640

`nextprime` — the next prime number . . . 641

`nops` — the number of operands . . . 643

`norm` — compute the norm of a matrix, a vector, or a polynomial . . . 645

`normal` — normalize an expression . . . 650

`nterms` — the number of terms of a polynomial . . . 653

`nthcoeff` — the n-th non-zero coefficient of a polynomial . . . 654

`nthmonomial` — the n-th monomial of a polynomial . . . 656

`nthterm` — the n-th term of a polynomial . . . 660

null — generate the void object of type DOM_NULL 662
numer — the numerator of a rational expression 665
ode — the domain of ordinary differential equations 666
op — the operands of an object 673
operator — define a new operator symbol 681
package — load a package of new library functions 684
pade — Pade approximation 693
partfrac — compute a partial fraction decomposition 696
patchlevel — the patch number of the installed MuPAD library 698
pathname — create a platform dependent path name 699
pdivide — pseudo-division of polynomials 701
piecewise — the domain of conditionally defined objects 703
plot — display graphical objects on the screen 718
plot2d — 2D plots 723
plot3d — 3D plots 733
plotfunc2d — 2D plots of function graphs 750
plotfunc3d — 3D plots of function graphs 761
plotOptions2d — scene options for 2D plots 767
plotOptions3d — scene options for 3D plots 778
point — generate a graphical point primitive 787
poly — create a polynomial 790
poly2list — convert a polynomial to a list of terms 798
polygon — generate a graphical polygon primitive 799
polylog — the polylogarithm function 809
powermod — compute a modular power of a number or a polynomial . . 812
print — print objects to the screen 815
proc — define a procedure 828
product — definite and indefinite products 839
protect — protect an identifier 841
protocol — create a protocol of a MuPAD session 844
psi — the digamma/polygamma function 846

quit — terminate the MuPAD session . . . 850
radsimp — simplify radicals in arithmetical expressions . . . 852
random — generate random integer numbers . . . 855
rationalize — transform an expression into a rational expression . . . 858
read — search, read, and execute a file . . . 859
readbytes, writebytes — read or write binary data from or to a file . 862
repeat, while — repeat and while loop . . . 869
rec — the domain of recurrence equations . . . 873
rectform — rectangular form of a complex expression . . . 876
register — remove the memory limit of the demo version . . . 884
reset — re-initialize a MuPAD session . . . 885
return — exit a procedure . . . 886
revert — revert lists or character strings, invert series expansions . . . 888
rewrite — rewrite an expression . . . 889
RGB — predefined color names . . . 892
save — save the state of an identifier . . . 893
select — select operands . . . 895
series — compute a (generalized) series expansion . . . 898
setuserinfo — set an information level . . . 915
share — create a unique data representation . . . 917
sign — the sign of a real or complex number . . . 918
signIm — the sign of the imaginary part of a complex number . . . 920
simplify — simplify an expression . . . 922
sin, cos, tan, csc, sec, cot — the trigonometric functions . . . 926
sinh, cosh, tanh, csch, sech, coth — the hyperbolic functions . . 931
slot — method or entry of a domain or a function environment . . . 936
solve — solve equations and inequalities . . . 943
solvers — an overview of MuPAD's solvers . . . 954
sort — sort a list . . . 958
sparsematrix — create a sparse matrix or a sparse vector . . . 960
split — split an object . . . 978

sqrt — the square root function 980
strmatch — match a pattern in a character string 983
subs — substitute into an object 985
subsex — extended substitution 991
subsop — replace operands 994
substring — extract a substring from a string 998
sum — definite and indefinite summation 1000
sysname — the name of the operating system 1004
sysorder — compare objects according to the internal order 1006
system — execute a command of the operating system 1008
table — create a table 1010
taylor — compute a Taylor series expansion 1013
tbl2text — concatenate the strings in a table 1017
tcoeff — the trailing coefficient of a polynomial 1017
testargs — decide whether procedure arguments should be tested . . . 1020
testtype — syntactical type checking 1022
text2expr — convert a character string to an expression 1025
text2int — convert a character string to an integer 1027
text2list, text2tbl — split a character string into substrings 1028
textinput — interactive input of text 1030
rtime, time — measure real time and CPU time 1032
traperror — trap errors 1035
type — the type of an object 1037
unassume — delete the properties of an identifier 1039
universe — the set-theoretical universe 1041
unloadmod — unload a module 1041
unprotect — remove protection of identifiers 1043
userinfo — print progress information 1044
val — the value of an object 1047
version — the version number of the MuPAD library 1048
warning — print a warning message 1049

`write` — write the values of variables into a file 1051

`zeta` — the Riemann zeta function 1054

`zip` — combine lists 1057

`stdlib`

In MuPAD, most of the provided functions are categorized into libraries and called with a prefix, such as `linalg::eigenvalues` or `numeric::eigenvalues`.

For convenience, a number of frequently used functions do not have such a prefix. These include the functions built into the MuPAD kernel such as the constructs of the MuPAD language. The current paper documents these basic functions.

`glossary` – glossary

This glossary explains some of the terms that are used throughout the MuPAD documentation.

arithmetical expression Syntactically, this is an object of `Type::Arithmetical`. In particular, this type includes numbers, identifiers and expressions of domain type `DOM_EXPR`.

domain The phrase "domain" is synonymous with "data type". Every MuPAD object has a data type referred to as its "domain type". It may be queried via the function `domtype`.
There are "basic domains" provided by the system kernel. These include various types of numbers, sets, lists, arrays, tables, expressions, polynomials etc. The documentation refers to these data types as "kernel domains". The name of the kernel domains are of the form `DOM_XXX` (e.g., `DOM_INT`, `DOM_SET`, `DOM_LIST`, `DOM_ARRAY`, `DOM_TABLE`, etc.). Details of kernel domains can be found in the document "Basic MuPAD Data Types".
In addition, MuPAD's programming language allows to introduce new data types via the keyword `domain` or the function `newDomain`. The MuPAD library provides many such domains. For example, series expansions, matrices, piecewise defined objects etc. are domains implemented in the MuPAD language. The documentation refers to such data types as "library domains". In particular, the library `Dom` provides a variety of predefined data types such as matrices, residue classes, intervals etc.
See `DOM_DOMAIN` for general explanations on data types. Here you also find some simple examples demonstrating how the user can implement her own domains.
For a concise description of MuPAD's domain concept, see the technical document "Axioms, Categories and Domains".

domain element The phrase "`x` is an element of the domain `d`" is synonymous with "`x` is of domain type `d`", i.e., "`domtype(x) = d`". In many cases, the help pages refer to "domain elements" as objects of library domains, i.e., the corresponding domain is implemented in the MuPAD language.

domain type The domain type of an object is the data type of the object. It may be queried via `domtype`.

flattening Sequences such as `a := (x, y)` or `b := (u, v)` consist of objects separated by commas. Several sequences may be combined to a longer sequence: `(a, b)` is "flattened" to the sequence `(x, y, u, v)` consisting of 4 elements. Most functions flatten their arguments, i.e., the call `f(a, b)` is interpreted as the call `f(x, y, u, v)` with 4 arguments. Note, however, that some functions (e.g., the operand function `op`) do not flatten their arguments: `op(a, 1)` is a call with 2 arguments.
The concept of flattening also applies to some functions such as `max`, where it refers to simplification rules such as `max(a, max(b, c)) = max(a, b, c)`.

function Typically, functions are represented by a procedure or a function environment. Also functional expressions such as `sin@exp + id^2`: $x \mapsto \sin(\exp(x)) + x^2$ represent functions. Also numbers can be used as (constant) functions. For example, the call `3(x)` yields the number `3` for any argument `x`.

number A number may be an integer (of type `DOM_INT`), or a rational number (of type `DOM_RAT`), or a real floating point number (of type `DOM_FLOAT`), or a complex number (of type `DOM_COMPLEX`).
The type `DOM_COMPLEX` encompasses the Gaussian integers such as `3 + 7*I`, the Gaussian rationals such as `3/4 + 7/4*I`, and complex floating point numbers such as `1.2 + 3.4*I`.

numerical expression This is an expression that does not contain any symbolic variable apart from the special constants `PI`, `E`, `EULER`, and `CATALAN`. A numerical expression such as `I^(1/3) + sqrt(PI)*exp(17)` is an exact representation of a real or a complex number; it may be composed of numbers, radicals and calls to special functions. It may be converted to a real or complex floating point number via `float`.

overloading The help page of a system function only documents the admissible arguments that are of some basic type provided by the MuPAD kernel. If the system function `f`, say, is declared as "overloadable", the user may extend its functionality. He can implement his own domain or function environment with a corresponding slot `"f"`. An element of this domain is then accepted by the system function `f` which calls the user-defined slot function.
See also the domain documentation.

polynomial	Syntactically, a polynomial such as `poly(x^2 + 2, [x])` is an object of type `DOM_POLY`. It must be created by a call to the function `poly`. Most functions that operate on such polynomials are overloaded by other polynomial domains of the MuPAD library.
polynomial expression	This is an arithmetical expression in which symbolic variables and combinations of such variables only enter via positive integer powers. Examples are `x^2 + 2` or `x*y + (z + 1)^2`.
rational expression	This is an arithmetical expression in which symbolic variables and combinations of such variables only enter via integer powers. Examples are `x^(-2) + x + 2` or `x*y + 1/(z + 1)^2`. Every polynomial expression is also a rational expression, but the two previous expressions are not polynomial expressions.

`mathematical constants and functions` – an overview

The following mathematical constants are predefined in MuPAD:

`complexInfinity`	–	complex infinity
`I`	–	imaginary unit $\sqrt{-1}$ (see `DOM_COMPLEX` for details)
`infinity`	–	real positive infinity
`undefined`	–	undefined value

The following constants are symbolic representations of special real numbers. Use `float` to get floating point approximations with the current precision `DIGITS`.

`CATALAN`	–	Catalan constant $\displaystyle\sum_{i=0}^{\infty} \frac{(-1)^i}{(2\,i+1)^2} = 0.9159..$
`E`, `exp(1)`	–	Euler number $\exp(1) = 2.718..$ (see `exp` for details)
`EULER`	–	Euler-Mascheroni constant $\displaystyle\lim_{n\to\infty}\left(\sum_{i=1}^{n}\frac{1}{i} - \ln(n)\right) = 0.5772..$
`PI`	–	$\pi = 3.141..$

The following mathematical functions are defined in a MuPAD session:

`abs`	–	absolute value of a real or complex number
`arg`	–	polar angle of a complex number
`bernoulli`	–	Bernoulli numbers and polynomials
`besselI`	–	modified Bessel functions of the first kind
`besselJ`	–	Bessel functions of the first kind
`besselK`	–	modified Bessel functions of the second kind
`besselY`	–	Bessel functions of the second kind
`beta`	–	beta function
`binomial`	–	binomial coefficient
`ceil`	–	nearest integer in the direction of ∞
`Ci`	–	cosine integral function
`dilog`	–	dilogarithm function
`dirac`	–	Dirac delta function
`Ei`	–	exponential integral function
`erf`	–	error function
`erfc`	–	complementary error function
`exp`	–	exponential function
`fact`	–	factorial function
`frac`	–	fractional part of a number
`floor`	–	nearest integer in the direction of $-\infty$
`gamma`	–	gamma function
`heaviside`	–	Heaviside step function
`hypergeom`	–	hypergeometric function
`igamma`	–	incomplete gamma function
`Im`	–	imaginary part of a complex number
`lambertV`	–	lower branch of Lambert's W function
`lambertW`	–	main branch of Lambert's W function
`log`	–	logarithm to an arbitrary base
`ln`	–	natural logarithm
`max`	–	maximum of real numbers
`min`	–	minimum of real numbers
`polylog`	–	polylogarithm function
`psi`	–	digamma/polygamma function
`Re`	–	real part of a complex number
`round`	–	rounding to the nearest integer
`Si`	–	sine integral function
`sign`	–	sign of a real or complex number
`sqrt`	–	square root function
`trunc`	–	nearest integer in the direction of 0
`zeta`	–	Riemann zeta function

Further, the trigonometric functions and hyperbolic functions

`cos`, `cot`, `csc`, `sec`, `sin`, `tan`, `cosh`, `coth`, `csch`, `sech`, `sinh`, `tanh`

and the inverse functions

`arccos`, `arccot`, `arccsc`, `arcsec`, `arcsin`, `arctan`, `arccosh`, `arccoth`, `arccsch` `arcsech`, `arcsinh`, `arctanh`

are implemented.

Changes:

- The hypergeometric functions `hypergeom` were implemented.

`options` – options used by MuPAD functions

The following options are used by MuPAD's system functions. These names are protected; they should not be assigned a value.

If a value is assigned to these names, the corresponding functions may respond to these options in an unexpected way.

The meaning of each option is documented on the help page of the corresponding function.

Option	used in
`Above`	`plot2d`, `plot3d`, `plotOptions2d`, `plotOptions3d`
`Adaptive`	`numeric::quadrature`
`All`	`anames`, `linalg::frobeniusForm`, `linalg::gaussElim`, `linalg::gaussJordan`, `linalg::hermiteForm`, `linalg::hessenberg`, `linalg::jordanForm`, `linopt::Transparent`, `linopt::corners`, `linopt::maximize`, `linopt::minimize`, `linopt::plot_data`, `lllint`, `prog::changes`, `prog::init`, `prog::testinit`
`Alldata`	`numeric::odesolve`, `numeric::odesolve2`, `numeric::odesolveGeometric`
`Always`	`Pref::keepOrder`
`AndMesh`	`plot3d`
`AndULine`	`plot3d`
`AndVLine`	`plot3d`
`Ansatz`	`detools::detSys`, `detools::ncDetSys`
`Any`	`setuserinfo`
`Append`	`fopen`, `fprint`, `prog::tcov`, `protocol`, `write`
`Approx`	`numeric::rationalize`
`Arch`	`sysname`
`Args`	`prog::calltree`
`Arrows`	`plotOptions2d`, `plotOptions3d`
`Ascii`	`plot2d`, `plot3d`, `plotOptions2d`, `plotOptions3d`

Option	used in
Ass	match
Assign	prog::check
Automatic	plotOptions2d, plotOptions3d
Autoreduced	detools::detSys, detools::ncDetSys
Averaged	stats::empiricalQuantile, stats::median
Axes	plotOptions2d, plotOptions3d
AxesInFront	plotOptions2d
AxesOrigin	plotOptions2d, plotOptions3d
AxesScaling	plotOptions2d, plotOptions3d
BUTCHER6	numeric::butcher, numeric::odesolve, numeric::odesolveGeometric
BackGround	plotOptions2d, plotOptions3d
BackSubstitution	solve
Backup	prog::trace
Banded	Dom::Matrix, Dom::MatrixGroup, Dom::SparseMatrix, Dom::SquareMatrix, matrix, sparsematrix
Below	plot2d, plot3d, plotOptions2d, plotOptions3d
BigEndian	readbytes, writebytes
Bin	fopen, fprint, write
Binary	operator, plot2d, plot3d, plotOptions2d, plotOptions3d
Box	plotOptions2d, plotOptions3d
BravaisPearson	stats::correlation
Byte	readbytes, writebytes
CDF	stats::csGOFT, stats::ksGOFT
CK45	numeric::butcher, numeric::odesolve, numeric::odesolveGeometric
CK54	numeric::butcher, numeric::odesolve, numeric::odesolveGeometric
CameraPoint	plotOptions3d
Capacity	Network::addEdge, Network::changeEdge, Network::new
Cartesian	linalg::laplacian, linalg::ogCoordTab
Center	output::tableForm, stringlib::format
Centers	plot::boxplot
ChangeOfVars	detools::transform
Circles	plot2d, plot3d, plotOptions2d, plotOptions3d
Closed	polygon
Coeffs	polylib::randpoly
CollectInformation	numlib::mpqs
Color	plot2d, plot3d, point, polygon

Option	used in
ColorPatches	plot3d
Colors	plot::boxplot
Column	output::tableForm
Comm	match
Complete	numeric::cubicSpline, numeric::cubicSpline2d
Cond	match, matchlib::analyze
Consecutive	listlib::sublist
Const	match, matchlib::analyze
Constrained	plot2d, plot3d, plotOptions2d, plotOptions3d
Continuous	int
Corner	plotOptions2d, plotOptions3d
Curve	plot2d, plot3d
Cyclic	text2list, text2tbl
Cylindrical	linalg::laplacian, linalg::ogCoordTab
DOPRI45	numeric::butcher, numeric::odesolve, numeric::odesolveGeometric
DOPRI54	numeric::butcher, numeric::odesolve, numeric::odesolveGeometric
DashedLines	plot2d, plot3d, plotOptions2d, plotOptions3d
DegInvLexOrder	degreevec, groebner::dimension, groebner::gbasis, groebner::normalf, groebner::spoly, lcoeff, lmonomial, lterm, nthcoeff, nthmonomial, nthterm, tcoeff
Degree	polylib::randpoly
DegreeOrder	degreevec, groebner::dimension, groebner::gbasis, groebner::normalf, groebner::spoly, lcoeff, lmonomial, lterm, nthcoeff, nthmonomial, nthterm, tcoeff
Delete	operator, prog::changes
Depth	prog::trace
Diagonal	Dom::Matrix, Dom::MatrixGroup, Dom::SparseMatrix, Dom::SquareMatrix, linalg::randomMatrix, matrix, sparsematrix
Diagonalization	numeric::expMatrix
Dimension	linalg::ogCoordTab
Discont	plot2d, plot3d, plotOptions2d
Domain	linsolve, prog::check, solve
DomainsOnly	Pref::keepOrder
DontRewriteBySystem	solve
Double	readbytes, writebytes
DrawMode	plot::boxplot

Option	used in
DualPrices	linopt::maximize, linopt::minimize
Duplicate	combinat::permutations, combinat::permute
EULER1	numeric::butcher, numeric::odesolve, numeric::odesolveGeometric
EllipticCylindrical	linalg::laplacian, linalg::ogCoordTab
Environment	prog::check
Error	prog::changes, unprotect
Escape	prog::check
Eweight	Network::addEdge, Network::changeEdge, Network::new
Exact	divide, numeric::rationalize
Expr	detools::detSys, detools::ncDetSys, gcd, interpolate, lcm, poly
Extended	Network::residualNetwork
Factor	numeric::polyroots, numeric::solve
Factorbase	numlib::mpqs
Fechner	stats::correlation
File	prog::tcov
Filled	polygon
FilledCircles	plot2d, plot3d, plotOptions2d, plotOptions3d
FilledSquares	plot2d, plot3d, plotOptions2d, plotOptions3d
First	stringlib::remove, stringlib::subs
FixedPrecision	numeric::polyroots, numeric::solve
Flat	plot2d, plot3d
Float	readbytes, writebytes
FocalPoint	plotOptions3d
Font	plotOptions2d, plotOptions3d
FontFamily	plotOptions2d, plotOptions3d
FontSize	plotOptions2d, plotOptions3d
FontStyle	plotOptions2d, plotOptions3d
Force	prog::trace, unloadmod
Forced	package
ForeGround	plotOptions2d, plotOptions3d
Frobenius	norm
Function	plot2d, plot3d
GC	numeric::quadrature
GL	numeric::quadrature
GT	numeric::quadrature
Gauss	numeric::quadrature
GaussChebyshev	numeric::quadrature
GaussLegendre	numeric::quadrature
GaussTschebyscheff	numeric::quadrature

Option	used in
Gif	plotOptions3d
Global	prog::check, unassume
Grid	plot2d, plot3d, plotOptions3d, plotfunc2d, plotfunc3d
GridLines	plot2d, plot3d, plotOptions2d
GridLinesColor	plot2d, plot3d, plotOptions2d
GridLinesStyle	plot2d, plot3d, plotOptions2d
GridLinesWidth	plot2d, plot3d, plotOptions2d
Hard	numeric::det, numeric::eigenvalues, numeric::eigenvectors, numeric::expMatrix, numeric::fMatrix, numeric::factorCholesky, numeric::factorLU, numeric::factorQR, numeric::fft, numeric::inverse, numeric::invfft, numeric::leastSquares, numeric::linsolve, numeric::matlinsolve, numeric::singularvalues, numeric::singularvectors
HardwareFloats	numeric::det, numeric::eigenvalues, numeric::eigenvectors, numeric::expMatrix, numeric::fMatrix, numeric::factorCholesky, numeric::factorLU, numeric::factorQR, numeric::fft, numeric::inverse, numeric::invfft, numeric::leastSquares, numeric::linsolve, numeric::matlinsolve, numeric::singularvalues, numeric::singularvectors
Height	plot2d, plot3d
HiddenLine	plot3d
Horizontal	plot::boxplot
Ident	matchlib::analyze
IgnoreProperties	solve
IgnoreSpecialCases	solve
Impulses	plot2d, plot3d
Include	linalg::expr2Matrix
Index	stringlib::contains, strmatch
IndexList	stringlib::contains
Infinity	norm
Info	prog::changes
Inner	combinat::compositions, combinat::integerVectors, combinat::partitions
InputOnly	protocol
IntMod	gcd, lcm, poly
Interactive	detools::detSys, detools::ncDetSys

Option	used in
InteractiveInput	numlib::mpqs
Interpolation	numeric::expMatrix
InverseTransformation	linalg::ogCoordTab
JPEG	plot2d, plot3d, plotOptions2d, plotOptions3d
KeepOrder	listlib::removeDuplicates, print
Krylov	numeric::expMatrix
Labeling	plotOptions2d, plotOptions3d
Labels	plotOptions2d, plotOptions3d
LargeFactorBound	numlib::mpqs
Laurent	Type::Series
Left	Series::Puiseux, Series::gseries, Type::Series, asympt, limit, output::tableForm, series, stringlib::format, student::plotRiemann, student::riemann
Length	Network::allShortPath, Network::longPath, Network::shortPath, Network::shortPathTo, combinat::compositions, combinat::integerVectors, combinat::partitions
Level	prog::check
LexOrder	degreevec, groebner::dimension, groebner::gbasis, groebner::normalf, groebner::spoly, lcoeff, lmonomial, lterm, nthcoeff, nthmonomial, nthterm, tcoeff
LieGroupAction	numeric::odesolveGeometric
Lin	plotOptions2d, plotOptions3d
LineStyle	plot2d, plot3d, plotOptions2d, plotOptions3d
LineWidth	plot2d, plot3d, plot::boxplot, plotOptions2d, plotOptions3d
Lines	plot2d, plot3d
LinesPoints	plot2d, plot3d
List	normal, plot2d, plot3d
LittleEndian	readbytes, writebytes
Local	prog::check
Localf	prog::check
Log	plotOptions2d, plotOptions3d
Logic	linopt::corners
MaxCalls	numeric::quadrature
MaxDegree	solve
MaxInFactorbase	numlib::mpqs

Option	used in
`MaxLength`	`combinat::compositions`, `combinat::integerVectors`, `combinat::partitions`
`MaxPart`	`combinat::compositions`, `combinat::integerVectors`, `combinat::partitions`
`MaxSlope`	`combinat::compositions`, `combinat::integerVectors`, `combinat::partitions`
`Mem`	`prog::trace`
`Merge`	`numeric::realroots`, `plot::boxplot`
`Mesh`	`plot3d`
`Middle`	`student::plotRiemann`, `student::riemann`
`MinLength`	`combinat::compositions`, `combinat::integerVectors`, `combinat::partitions`
`MinPart`	`combinat::compositions`, `combinat::integerVectors`, `combinat::partitions`
`MinSlope`	`combinat::compositions`, `combinat::integerVectors`, `combinat::partitions`
`Minimize`	`numeric::rationalize`
`MinorExpansion`	`numeric::det`
`Mode`	`plot2d`, `plot3d`
`Monic`	`groebner::gbasis`
`MultiSolutions`	`numeric::fsolve`, `numeric::solve`
`Multiple`	`linalg::eigenvalues`, `numeric::solve`, `solve`
`NC`	`numeric::quadrature`
`Name`	`setuserinfo`
`Nary`	`operator`
`Natural`	`numeric::cubicSpline`, `numeric::cubicSpline2d`
`NewVars`	`detools::transform`
`NewtonCotes`	`numeric::quadrature`
`NoArgs`	`prog::trace`
`NoCheck`	`linalg::factorCholesky`, `numeric::factorCholesky`
`NoErrors`	`numeric::eigenvectors`, `numeric::singularvectors`
`NoLeftVectors`	`numeric::eigenvectors`, `numeric::singularvectors`
`NoNL`	`fprint`, `print`, `userinfo`
`NoOperators`	`misc::maprec`

Option	used in
NoRightVectors	numeric::eigenvectors, numeric::singularvectors
NoWarning	numeric::inverse, numeric::leastSquares, numeric::linsolve, numeric::matlinsolve, series, stats::equiprobableCells
NonNegative	linopt::Transparent, linopt::corners, linopt::maximize, linopt::minimize, linopt::plot_data
NonNested	import::readdata
None	plotOptions2d, plotOptions3d, unprotect
Normal	stats::tTest
Not	stats::selectRow
NotAKnot	numeric::cubicSpline, numeric::cubicSpline2d
Notched	plot::boxplot
Null	match
NumberOfPolynomials	numlib::mpqs
Off	plotOptions2d, plotOptions3d
On	plotOptions2d, plotOptions3d
Only	plot3d
Order	groebner::gbasis
Origin	plotOptions2d, plotOptions3d
Outer	combinat::compositions, combinat::integerVectors, combinat::partitions
Output	output::tableForm
PDF	stats::csGOFT
PF	stats::csGOFT
ParabolicCylindrical	linalg::laplacian, linalg::ogCoordTab
Param	detools::detSys, detools::ncDetSys
Path	Network::allShortPath, Network::longPath, Network::shortPath, Network::shortPathTo, prog::tcov
Periodic	numeric::cubicSpline, numeric::cubicSpline2d
Plain	fread, operator, prog::calltree, prog::trace, read
PlotDevice	plot2d, plot3d, plotOptions2d, plotOptions3d
PointStyle	plot2d, plot3d, plotOptions2d, plotOptions3d
PointWidth	plot2d, plot3d, plotOptions2d, plotOptions3d
Points	plot2d, plot3d

Option	used in
PolyExpr	indets
Population	stats::covariance, stats::stdev, stats::variance
PostMap	misc::maprec
Postfix	operator
Postscript	plot2d, plot3d, plotOptions2d, plotOptions3d
PreMap	misc::maprec
Prefix	operator
Pretty	userinfo
PrimeLimit	ifactor
PrincipalValue	int, solve
Properties	anames
Protect	prog::check
ProtectLevelError	protect, unprotect
ProtectLevelNone	protect, unprotect
ProtectLevelWarning	protect, unprotect
Protected	anames
Puiseux	Type::Series
QRD	numeric::leastSquares
Quiet	fread, package, prog::changes, prog::exprtree, read, setuserinfo
Quo	divide, pdivide
RK4	numeric::butcher, numeric::odesolve, numeric::odesolveGeometric
RKF34	numeric::butcher, numeric::odesolve, numeric::odesolveGeometric
RKF43	numeric::butcher, numeric::odesolve, numeric::odesolveGeometric
RKF45a	numeric::butcher, numeric::odesolve, numeric::odesolveGeometric
RKF45b	numeric::butcher, numeric::odesolve, numeric::odesolveGeometric
RKF54a	numeric::butcher, numeric::odesolve, numeric::odesolveGeometric
RKF54b	numeric::butcher, numeric::odesolve, numeric::odesolveGeometric
RKF78	numeric::butcher, numeric::odesolve, numeric::odesolveGeometric
RKF87	numeric::butcher, numeric::odesolve, numeric::odesolveGeometric
Random	numeric::fsolve, numeric::solve
Ranges	linalg::ogCoordTab
Raster	plotOptions2d, plotOptions3d

Option	used in
RatExpr	indets
Raw	fopen, readbytes, writebytes
Read	fopen
Real	Series::Puiseux, Type::Series, limit, series
RealValuesOnly	plot2d, plot3d, plotOptions2d, plotOptions3d
RealsOnly	plotOptions2d, plotOptions3d
Recursive	prog::allFunctions
RelativeError	numeric::odesolve, numeric::odesolve2, numeric::odesolveGeometric
Rem	divide, lmonomial, pdivide
Remove	prog::changes
Reorder	groebner::gbasis
Repeat	combinat::subwords
Restore	numeric::rationalize
RestrictedSearch	numeric::fsolve, numeric::solve
ReturnType	numeric::eigenvectors, numeric::expMatrix, numeric::fMatrix, numeric::factorCholesky, numeric::factorLU, numeric::factorQR, numeric::inverse, numeric::leastSquares, numeric::matlinsolve, numeric::singularvectors
Right	Series::Puiseux, Series::gseries, Type::Series, asympt, limit, output::tableForm, series, stringlib::format, student::plotRiemann, student::riemann
Root	pathname
RotationParabolic	linalg::laplacian, linalg::ogCoordTab
SVD	numeric::leastSquares
Sample	stats::covariance, stats::stdev, stats::variance
Save	prog::check
Scales	linalg::ogCoordTab
Scaling	plot2d, plot3d, plotOptions2d, plotOptions3d
Screen	plotOptions2d, plotOptions3d
SearchLevel	numeric::realroot

Option	used in
`Seed`	`stats::betaRandom`, `stats::binomialRandom`, `stats::cauchyRandom`, `stats::chisquareRandom`, `stats::erlangRandom`, `stats::exponentialRandom`, `stats::fRandom`, `stats::gammaRandom`, `stats::geometricRandom`, `stats::hypergeometricRandom`, `stats::logisticRandom`, `stats::normalRandom`, `stats::poissonRandom`, `stats::tRandom`, `stats::uniformRandom`, `stats::weibullRandom`
`Shift`	`plot::boxplot`
`Short`	`readbytes`, `writebytes`
`ShowAssumptions`	`linsolve`, `numeric::linsolve`, `numeric::matlinsolve`
`SieveArrayLimit`	`numlib::mpqs`
`SignedByte`	`readbytes`, `writebytes`
`SignedShort`	`readbytes`, `writebytes`
`SignedWord`	`readbytes`, `writebytes`
`Small`	`output::tree`
`Smoothness`	`plot2d`, `plot3d`
`Soft`	`numeric::det`, `numeric::eigenvalues`, `numeric::eigenvectors`, `numeric::expMatrix`, `numeric::fMatrix`, `numeric::factorCholesky`, `numeric::factorLU`, `numeric::factorQR`, `numeric::fft`, `numeric::inverse`, `numeric::invfft`, `numeric::leastSquares`, `numeric::linsolve`, `numeric::matlinsolve`, `numeric::singularvalues`, `numeric::singularvectors`
`SoftwareFloats`	`numeric::det`, `numeric::eigenvalues`, `numeric::eigenvectors`, `numeric::expMatrix`, `numeric::fMatrix`, `numeric::factorCholesky`, `numeric::factorLU`, `numeric::factorQR`, `numeric::fft`, `numeric::inverse`, `numeric::invfft`, `numeric::leastSquares`, `numeric::linsolve`, `numeric::matlinsolve`, `numeric::singularvalues`, `numeric::singularvectors`
`SolidLines`	`plot2d`, `plot3d`, `plotOptions2d`, `plotOptions3d`
`Solved`	`detools::detSys`
`Sort`	`output::tableForm`

Option	used in
Special	linalg::matlinsolve, prog::check
Spherical	linalg::laplacian, linalg::ogCoordTab
SquareFree	numeric::polyroots, numeric::solve
Squares	plot2d, plot3d, plotOptions2d, plotOptions3d
StartingValues	stats::reg
Stat	prog::tcov
Status	prog::test
Steps	detools::ncDetSys, plotOptions2d, plotOptions3d
Stepsize	numeric::odesolve, numeric::odesolve2, numeric::odesolveGeometric
Style	plot2d, plot3d
Symbolic	detools::modode, linsolve, numeric::cubicSpline, numeric::cubicSpline2d, numeric::det, numeric::factorCholesky, numeric::factorLU, numeric::factorQR, numeric::fft, numeric::inverse, numeric::invfft, numeric::leastSquares, numeric::linsolve, numeric::matlinsolve, numeric::odesolve, numeric::odesolve2, numeric::odesolveGeometric, stats::tCDF
Symmetric	numeric::factorCholesky
System	Pref::keepOrder
TIFF	plot2d, plot3d, plotOptions2d, plotOptions3d
Table	Dom::BaseDomain
Taylor	Type::Series
TaylorExpansion	numeric::expMatrix
TempFile	fopen, readbytes, writebytes
Terms	polylib::randpoly
Test	linalg::vectorPotential
Text	finput, fopen, fprint, userinfo, write
Ticks	plot2d, plot3d, plotOptions2d, plotOptions3d
Title	plot2d, plot3d, plotOptions2d, plotOptions3d
TitlePosition	plot2d, plot3d, plotOptions2d, plotOptions3d
Tolerance	numlib::mpqs
Torus	linalg::laplacian, linalg::ogCoordTab
Transformation	linalg::ogCoordTab
Transparent	plot3d

Option	used in
Transposed	linalg::vandermondeSolve
TrapError	prog::test
Tree	Dom::BaseDomain, prog::calltree
ULine	plot3d
UnConstrained	plotOptions2d, plotOptions3d
Undefined	discont
Undirected	Series::Puiseux, Type::Series, series
Unimodular	linalg::randomMatrix
Unique	linalg::matlinsolve, output::tableForm
UnitVectors	linalg::ogCoordTab
Unquoted	fprint, output::tableForm, print
UnrestrictedSearch	numeric::fsolve, numeric::solve
Unsimplified	subs, subsex, subsop
UsePrimeTab	ifactor
User	anames
VLine	plot3d
Vertical	plot::boxplot
ViewingBox	plot2d, plot3d, plotOptions2d, plotOptions3d
Vweight	Network::addVertex, Network::changeVertex, Network::new
Warning	prog::changes, unprotect
Width	output::tableForm
Widths	plot::boxplot
WireFrame	plot3d
Word	readbytes, writebytes
Write	fopen, fprint, protocol, write
XMax	plotOptions2d, plotOptions3d
XMin	plotOptions2d, plotOptions3d
YMax	plotOptions2d, plotOptions3d
YMin	plotOptions2d, plotOptions3d
ZMax	plotOptions3d
ZMin	plotOptions3d
andor	rewrite
dom	slot
escape	proc
hold	proc
logic	simplify
noDebug	debug, proc
relation	simplify
remember	proc
sincos	combine, rewrite
sinhcosh	combine, rewrite
xCK45	numeric::odesolve, numeric::odesolveGeometric

Option	used in
`xCK54`	`numeric::odesolve`, `numeric::odesolveGeometric`
`xDOPRI45`	`numeric::odesolve`, `numeric::odesolveGeometric`
`xDOPRI54`	`numeric::odesolve`, `numeric::odesolveGeometric`
`xRKF34`	`numeric::odesolve`, `numeric::odesolveGeometric`
`xRKF43`	`numeric::odesolve`, `numeric::odesolveGeometric`
`xRKF45a`	`numeric::odesolve`, `numeric::odesolveGeometric`
`xRKF45b`	`numeric::odesolve`, `numeric::odesolveGeometric`
`xRKF54a`	`numeric::odesolve`, `numeric::odesolveGeometric`
`xRKF54b`	`numeric::odesolve`, `numeric::odesolveGeometric`
`xRKF78`	`numeric::odesolve`, `numeric::odesolveGeometric`
`xRKF87`	`numeric::odesolve`, `numeric::odesolveGeometric`

`:=` – assign variables

`x := value` assigns the variable `x` a `value`.

`[x1, x2, ...] := [value1, value2, ...]` assigns the variables `x1`, `x2` etc. the corresponding values `value1`, `value2` etc.

`f(X1, X2, ...) := value` adds an entry to the remember table of the procedure `f`.

Call(s):

- `x := value`
- `_assign(x, value)`
- `[x1, x2, ...]  := [value1, value2, ...]`
- `_assign([x1, x2, ...], [value1, value2, ...])`
- `f(X1, X2, ...)  := value`
- `_assign(f(X1, X2, ...), value)`

Parameters:

`x, x1, x2, ...`	—	identifiers or indexed identifiers
`value, value1, value2, ...`	—	arbitrary MuPAD objects
`f`	—	a procedure or a function environment
`X1, X2, ...`	—	arbitrary MuPAD objects

Return Value: `value` or `[value1, value2, ...]`, respectively.

Related Functions: `anames`, `assign`, `assignElements`, `delete`, `evalassign`

Details:

- `_assign(x, value)` is equivalent to `x := value`.
- `_assign([x1, x2, ...], [value1, value2, ...])` is equivalent to `[x1, x2, ...] := [value1, value2, ...]`. Both lists must have the same number of elements.
- If `x` is neither a list, nor a table, nor an array, nor a matrix, nor an element of a domain with a slot `"set_index"`, then an indexed assignment such as `x[i] := value` implicitly turns the identifier `x` into a table with a single entry `(i = value)`. Cf. example 2.
- The assignment `f(X1, X2, ...) := value` adds an entry to the remember table of the procedure `f`.

  If `f` is neither procedure nor a function environment, then `f` is implicitly turned into a (trivial) procedure with a single entry `(X1, X2, ...) = value` in its remember table. Cf. example 3.
- Identifiers on the left hand side of an assignment are not evaluated (use `evalassign` if this is not desired). I.e., in `x := value`, the previous value of `x`, if any, is deleted and replaced by the new value. Note, however, that the index of an indexed identifier is evaluated. I.e., in `x[i] := value`, the index `i` is replaced by its current value before the corresponding entry of `x` is assigned the value. Cf. example 4.
- `_assign` is a function of the system kernel.

Example 1. The assignment operator `:=` can be applied to a single identifier as well as to a list of identifiers:

```
>> x := 42: [x1, x2, x3] := [43, 44, 45]: x, x1, x2, x3

                        42, 43, 44, 45
```

In case of lists, all variables of the left-hand side are assigned their values *simultaneously*:

```
>> [x1, x2] := [3, 4]: [x1, x2] := [x2, x1]: x1, x2

                              4, 3
```

The functional equivalent of the assign operator `:=` is the function `_assign`:

```
>> _assign(x, 13): _assign([x1, x2], [14, 15]): x, x1, x2

                            13, 14, 15
```

Assigned values are deleted via the keyword `delete`:

```
>> delete x, x1, x2: x, x1, x2

                             x, x1, x2
```

Example 2. Assigning a value to an indexed identifier, a corresponding table (`table`, `DOM_TABLE`) is generated implicitly, if the identifier was not assigned a list, a table, an array, or a matrix before:

```
>> delete x: x[1] := 7: x

                              table(
                                1 = 7
                              )
```

If `x` is a list, a table, an array, or a matrix, then an indexed assignment adds a further entry or changes an existing entry:

```
>> x[abc] := 8: x

                            table(
                              abc = 8,
                              1 = 7
                            )
```

```
>> x := [a, b, c, d]: x[3] := new: x

                          [a, b, new, d]
```

```
>> x := array(1..2, 1..2): x[2, 1] := value: x

                      +-                    -+
                      |  ?[1, 1], ?[1, 2]   |
                      |                      |
                      |   value,  ?[2, 2]   |
                      +-                    -+
```

```
>> delete x:
```

Example 3. Consider a simple procedure:

```
>> f := x -> sin(x)/x: f(0)

 Error: Division by zero;
 during evaluation of 'f'
```

The following assignment adds an entry to the remember table:

```
>> f(0) := 1: f(0)

                                   1
```

If f does not evaluate to a function, then a trivial procedure with a remember table is created implicitly:

```
>> delete f: f(x) := x^2: expose(f)

                       proc()
                         name f;
                         option remember;
                       begin
                         procname(args())
                       end_proc
```

Note that the remember table only provides a result for the input x:

```
>> f(x), f(1.0*x), f(y)

                        2
                       x , f(1.0 x), f(y)

>> delete f:
```

Example 4. The left hand side of an assignment is not evaluated. In the following, x := 3 assigns a new value to x, not to y:

```
>> x := y: x := 3: x, y

                                3, y
```

Consequently, the following is not a multiple assignment to the identifiers in the list, but a single assignment to the list L:

```
>> L := [x1, x2]: L := [21, 22]: L, x1, x2

                           [21, 22], x1, x2
```

However, indices are evaluated in indexed assignments:

```
>> i := 2: x[i] := value: x
```

```
table(
  2 = value
)
```

```
>> for i from 1 to 3 do x[i] := i^2 end_for: x
```

```
table(
  3 = 9,
  1 = 1,
  2 = 4
)
```

```
>> delete x, L, i:
```

Example 5. Since an assignment has a return value (the assigned value), the following command assigns values to several identifiers simultaneously:

```
>> a := b := c := 42: a, b, c
```

```
42, 42, 42
```

For syntactical reasons, the inner assignment has to be enclosed by additional brackets in the following command:

```
>> a := sin((b := 3)): a, b
```

```
sin(3), 3
```

```
>> delete a, b, c:
```

. – concatenate objects

`object1.object2` concatenates two objects.

`_concat(object1, object2, ...)` concatenates an arbitrary number of objects.

Call(s):

- `object1 . object2`
- `_concat(object1, object2, ...)`

Parameters:

`object1` — a character string, an identifier, or a list
`object2` — a character string, an identifier, an integer, or a list

Return Value: an object of the same type as `object1`.

Overloadable by: `object1, object2, ...`

Related Functions: `@`, `append`

Details:

- `_concat(object1, object2)` is equivalent to `object1.object2`. The function call `_concat(object1, object2, object3, ...)` is equivalent to

  `((object1.object2).object3). ... .`

  `_concat()` returns the void object of type `DOM_NULL`.

- The following combinations are possible:

`object1`	`object2`	`object1.object2`
string	string	string
string	identifier	string
string	integer	string
identifier	string	identifier
identifier	identifier	identifier
identifier	integer	identifier
list	list	list

  E.g., `x.1` creates the identifier `x1`.

- Note that the objects to be concatenated are evaluated before concatenation. Thus, if `x := y`, `i := 1`, the concatenation `x.i` produces the identifier `y1`. However, the resulting identifier `y1` is *not* fully evaluated. Cf. example 2.

- `_concat` is a function of the system kernel.

Example 1. We demonstrate all possible combinations of types that can be concatenated. Strings are produced if the first object is a string:

```
>> "x"."1", "x".y, "x".1

                    "x1", "xy", "x1"
```

Identifiers are produced if the first object is an identifier:

```
>> x."1", x.y , x.1
```

```
x1, xy, x1
```

The concatenation operator . also serves for concatenating lists:

```
>> [1, 2] . [3, 4]
```

```
[1, 2, 3, 4]
```

```
>> L := []: for i from 1 to 10 do L := L . [x.i] end_for: L
```

```
[x1, x2, x3, x4, x5, x6, x7, x8, x9, x10]
```

```
>> delete L:
```

Example 2. We demonstrate the evaluation strategy of concatenation. Before concatenation, the objects are evaluated:

```
>> x := "Val": i := ue: x.i
```

```
"Value"
```

```
>> ue := 1:  x.i
```

```
"Val1"
```

An identifier produced via concatenation is not fully evaluated:

```
>> delete x: x1 := 17: x.1, eval(x.1)
```

```
x1, 17
```

The . operator can be used to create variables dynamically. They can be assigned values immediately:

```
>> delete x: for i from 1 to 5 do x.i := i^2 end_for:
```

Again, the result of the concatenation is not fully evaluated:

```
>> x.i $ i= 1..5
```

```
x1, x2, x3, x4, x5
```

```
>> eval(%)
```

```
1, 4, 9, 16, 25
```

```
>> delete i, ue: (delete x.i) $ i = 1..5:
```

Example 3. The function `_concat` can be used to concatenate an arbitrary number of objects:

```
>> _concat("an", " ", "ex", "am", "ple")

                          "an example"

>> _concat("0", " ".i $ i = 1..15)

             "0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15"

>> _concat([], [x.i] $ i = 1..10)

            [x1, x2, x3, x4, x5, x6, x7, x8, x9, x10]
```

`..` – range operator

`l..r` defines a “range” with the left bound `l` and the right bound `r`.

Call(s):

- `l .. r`
- `_range(l, r)`

Parameters:

`l, r` — arbitrary MuPAD objects

Return Value: an expression of type `"_range"`.

Overloadable by: `l, r`

Related Functions: `$`, `Dom::Interval`

Details:

- A range is a technical construct that is used to specify ranges of numbers when calling various system functions such as `int`, `array`, `op`, or the sequence operator `$`. Usually, `l..r` represents a real interval (e.g., `int(f(x), x = l..r)`), or the sequence of integers from `l` to `r`.
- `_range(l, r)` is equivalent to `l..r`.
- To create and operate on intervals in a mathematical sense, use the data type `Dom::Interval`.
- `_range` is a function of the system kernel.

Example 1. A range can be defined with the `..` operator as well as with a call to the function `_range`:

```
>> _range(1, 42), 1..42

                        1..42, 1..42
```

In the following call, the range represents an interval:

```
>> int(x, x = l..r)

                            2    2
                           r    l
                           -- - --
                            2    2
```

Ranges can be used for accessing the operands of expressions or to define the dimension of an array:

```
>> op(f(a, b, c, d, e), 2..4)

                           b, c, d

>> array(1..3, [a1, a2,a3])

                        +-           -+
                        | a1, a2, a3 |
                        +-           -+
```

Ranges can also be used for creating expression sequences:

```
>> i^3 $ i = 1..5

                       1, 8, 27, 64, 125
```

Example 2. The range operator `..` is a technical device that does not check its parameters with respect to their semantics. It just creates a range which is interpreted in the context in which it is used later. Any bounds are accepted:

```
>> float(PI) .. -sqrt(2)/3

                                  1/2
                                 2
                      3.141592654..- ----
                                   3
```

`=`, `<>` – equations and inequalities

`x = y` defines an equation.

`x <> y` defines an inequality.

Call(s):

- `x = y`
- `_equal(x, y)`
- `x <> y`
- `_unequal(x, y)`

Parameters:

`x, y` — arbitrary MuPAD objects

Return Value: an expression of type `"_equal"` or `"_unequal"`, respectively.

Related Functions: `<`, `<=`, `>`, `>=`, `and`, `bool`, `FALSE`, `if`, `lhs`, `not`, `or`, `repeat`, `rhs`, `solve`, `TRUE`, `while`, `UNKNOWN`

Details:

- `x = y` is equivalent to the function call `_equal(x, y)`.
- `x <> y` is equivalent to the function call `_unequal(x, y)`.
- The operators `=` and `<>` return symbolic expressions representing equations and inequalities, respectively.

  The resulting expressions can be evaluated to `TRUE` or `FALSE` by the function `bool`. They also serve as control conditions in `if`, `repeat`, and `while` statements. In all these cases, testing for equality or inequality is a purely syntactical test. E.g., `bool(0.5 = 1/2)` returns `FALSE` although both numbers coincide numerically. Correspondingly, `bool(0.5 <> 1/2)` returns `TRUE`.

  Further, Boolean expressions can be evaluated to `TRUE`, `FALSE`, or `UNKNOWN` by the function `is`. Tests using `is` are semantical comparing `x` and `y` subject to mathematical considerations.
- Equations and inequalities have two operands: the left hand side and the right hand side. One may use `lhs` and `rhs` to extract these operands.
- `not x = y` is always converted to `x <> y`.
- `not x <> y` is always converted to `x = y`.
- `_equal` is a function of the system kernel.
- `_unequal` is a function of the system kernel.

Example 1. In the following, note the difference between syntactical and numerical equality. The numbers 1.5 and 3/2 coincide numerically. However, 1.5 is of domain type `DOM_FLOAT`, whereas 3/2 is of domain type `DOM_RAT`. Consequently, they are not regarded as equal in the following syntactical test:

```
>> 1.5 = 3/2; bool(%)

                              1.5 = 3/2

                                FALSE
```

The following expressions coincide syntactically:

```
>> _equal(1/x, diff(ln(x),x)); bool(%)

                                1   1
                                - = -
                                x   x

                                TRUE
```

The Boolean operator `not` converts equalities and inequalities:

```
>> not a = b, not a <> b

                             a <> b, a = b
```

Example 2. The examples below demonstrate how `=` and `<>` deal with non-mathematical objects and data structures:

```
>> if "text" = "t"."e"."x"."t" then "yes" else "no" end

                                "yes"

>> bool(table(a = PI) <> table(a = sqrt(2)))

                                TRUE
```

Example 3. We demonstrate the difference between the syntactical test via `bool` and the semantical test via `is`:

```
>> bool(1 = x/(x + y) + y/(x + y)), is(1 = x/(x + y) + y/(x + y))

                             FALSE, TRUE
```

Example 4. Equations and inequalities are typical input objects for system functions such as `solve`:

```
>> solve(x^2 - 2*x = -1, x)

                              {1}

>> solve(x^2 - 2*x <> -1, x)

                          C_ minus {1}
```

`<, <=, >, >=` – inequalities

`x < y`, `x <= y`, `x > y`, and `x >= y` define inequalities.

Call(s):

- `x < y`
- `_less(x, y)`
- `x <= y`
- `_leequal(x, y)`
- `x > y`
- `_less(y, x)`
- `x >= y`
- `_leequal(y, x)`

Parameters:

`x, y` — arbitrary MuPAD objects

Return Value: an expression of type `"_less"` or `"_leequal""`, respectively.

Overloadable by: `x, y`

Related Functions: `<>`, `=`, `and`, `bool`, `FALSE`, `if`, `lhs`, `not`, `or`, `repeat`, `rhs`, `solve`, `TRUE`, `while`, `UNKNOWN`

Details:

- `x > y` and `x >= y` are always converted to `y < x` and `y <= x`, respectively.
- `x < y` is equivalent to the function call `_less(x,y)`. It represents the Boolean statement "`x` is smaller than `y`".

- `x <= y` is equivalent to the function call `_leequal(x,y)`. It represents the Boolean statement "`x` is smaller than or equal to `y`".
- These operators return symbolic Boolean expressions. If only real numbers of `Type::Real` are involved, these expressions can be evaluated to `TRUE` or `FALSE` by the function `bool`. They also serve as control conditions in `if`, `repeat`, and `while` statements. For floating point intervals, these operators are interpreted as "strictly smaller than" and so on, see example 2.

  Further, Boolean expressions can be evaluated to `TRUE`, `FALSE`, or `UNKNOWN` by the function `is`. Tests using `is` can also be applied to constant symbolic expressions. Cf. example 4.
- `bool` also handles inequalities involving character strings. It compares them with respect to the lexicographical ordering.
- Inequalities have two operands: the left hand side and the right hand side. One may use `lhs` and `rhs` to extract these operands.
- `_less` is a function of the system kernel.
- `_leequal` is a function of the system kernel.

Example 1. The operators `<`, `<=`, `>`, and `>=` produce symbolic inequalities. They can be evaluated to `TRUE` or `FALSE` by the function `bool` if only real numbers of type `Type::Real` (integers, rationals, and floats) are involved:

```
>> 1.5 <= 3/2; bool(%)

                          1.5 <= 3/2

                             TRUE
```

Note that `bool` does not handle Boolean expressions that involve exact expressions, even if they represent real numbers:

```
>> _less(PI,  sqrt(2) + 17/10); bool(%)

                             1/2
                        PI < 2    + 17/10
 Error: Can't evaluate to boolean [_less]
```

Example 2. Comparison of intervals is interpreted as "strict", that is, all combinations of numbers in the intervals must fulfill the relation:

```
>> bool(0...1 < 2...3), bool(0...2 < 1...3),
   bool(0...1 < 1...2)
```

```
                    TRUE, FALSE, FALSE
```

```
>> bool(0...1 <= 2...3), bool(0...2 <= 1...3),
   bool(0...1 <= 1...2)
```

```
                    TRUE, FALSE, TRUE
```

Example 3. This examples demonstrates how character strings can be compared:

```
>> if "text" < "t"."e"."x"."t"."book" then "yes" else "no" end
```

```
                         "yes"
```

```
>> bool("aa" >= "b")
```

```
                         FALSE
```

Example 4. Note that `bool` only compares numbers of type `Type::Real`, whereas `is` can also compare exact constant expressions:

```
>> bool(10 < PI^2 + sqrt(2)/10)
```

```
 Error: Can't evaluate to boolean [_less]
```

```
>> is(10 < PI^2 + sqrt(2)/10)
```

```
                         TRUE
```

Example 5. Inequalities are valid input objects for the system function `solve`:

```
>> solve(x^2 - 2*x < 3, x)
```

```
                        ]-1, 3[
```

```
>> solve(x^2 - 2*x >= 3, x)
```

```
              ]-infinity, -1] union [3, infinity[
```

Example 6. The operators `<` and `<=` can be overloaded by user-defined domains:

```
>> myDom := newDomain("myDom"): myDom::print := x -> extop(x):
```

Without overloading `_less` or `_leequal`, elements of this domain cannot be compared:

```
>> x := new(myDom, PI): y := new(myDom, sqrt(10)): bool(x < y)
```

```
 Error: Can't evaluate to boolean [_less]
```

Now, a slot `"_less"` is defined. It is called, when an inequality of type `"_less"` is evaluated by `bool`. The slot compares floating point approximations if the arguments are not of type `Type::Real`:

```
>> myDom::_less := proc(x, y)
   begin
        x := extop(x, 1):
        y := extop(y, 1):
        if not testtype(x, Type::Real) then
           x := float(x):
           if not testtype(x, Type::Real) then
              error("cannot compare")
           end_if
        end_if:
        if not testtype(y, Type::Real) then
           y := float(y):
           if not testtype(y, Type::Real) then
              error("cannot compare")
           end_if
        end_if:
        bool(x < y)
   end_proc:
```

```
>> x, y, bool(x < y), bool(x > y)
```

```
                          1/2
                  PI, 10   , TRUE, FALSE
```

```
>> bool(new(myDom, I) < new(myDom, PI))
```

```
 Error: cannot compare [myDom::_less]
```

```
>> delete myDom, x, y:
```

+ – add expressions

`x + y + ...` computes the sum of `x`, `y` etc.

Call(s):

- `x + y + ...`
- `_plus(x, y, ...)`

Parameters:

`x, y, ...` — arithmetical expressions, polynomials of type `DOM_POLY`, sets, equations, inequalities, or comparisons

Return Value: an arithmetical expression, a polynomial, a set, an equation, an inequality, or a comparison.

Overloadable by: `x, y, ...`

Related Functions: `^`, `/`, `*`, `-`, `_invert`, `_negate`, `poly`, `Pref::keepOrder`, `sum`

Details:

- `x + y + ...` is equivalent to the function call `_plus(x, y, ...)`.
- All terms that are numbers of type `Type::Numeric` are automatically combined to a single number.
- Terms of a symbolic sum may be rearranged internally. Cf. example 1. The user can control the ordering by the preference `Pref::keepOrder`. See also the documentation for `print`.
- `_plus` accepts an arbitrary number of arguments. In conjunction with the sequence operator `$`, this function is the recommended tool for computing finite sums. Cf. example 2. The function `sum` may also serve for computing such sums. However, `sum` is designed for the computation of symbolic and infinite sums. It is slower than `_plus`.
- `x - y` is internally represented as `x + y*(-1)` = `_plus(x, _mult(y, -1))`. See `_subtract` for details.
- For adding equalities, inequalities, and comparisons, the following rules are implemented:

    - Adding an arithmetical expression adds the expression to both sides.
    - Adding an equality adds the left hand sides and the right hand sides separately.
    - Adding a comparison does likewise, taking care of the correct operator. Adding a comparison to an inequality is not permitted.

  Cf. example 4.

- Many library domains overload `_plus` by an appropriate slot `"_plus"`. Sums involving elements of library domains are processed as follows:

  A sum `x + y + ...` is searched for elements of library domains from left to right. Let `z` be the first term that is not of one of the basic types provided by the kernel (numbers, expressions, etc.). If the domain `d` = `z::dom` = `domtype(z)` has a slot `"_plus"`, it is called in the form `d::_plus(x, y, ...)`. The result returned by `d::_plus` is the result of `x + y + ...`.

  Users should implement the slot `d::_plus` of their domains `d` according to the following convention:

  - If all terms are elements of `d`, an appropriate sum of type `d` should be returned.
  - If at least one term cannot be converted to an element of `d`, the slot should return `FAIL`.
  - Care must be taken if there are terms that are not of type `d`, but can be converted to type `d`. Such terms should be converted only if the mathematical semantics is obvious to any user who uses this domain as a 'black box' (e.g., integers may be regarded as rational numbers because of the natural mathematical embedding). If in doubt, the `"_plus"` method should return `FAIL` instead of using implicit conversions. If implicit conversions are used, they must be well-documented.

  Cf. examples 6 and 7.

  Most of the library domains in MuPAD's standard installation comply with this convention.

- `_plus()` returns the number 0.
- Polynomials of type `DOM_POLY` are added by `+`, if they have the same indeterminates and the same coefficient ring.
- For finite sets `X`, `Y`, the sum `X + Y` is the set $\{x + y;\ x \in X,\ y \in X\}$.
- `_plus` is a function of the system kernel.

Example 1. Numerical terms are simplified automatically:

```
>> 3 + x + y + 2*x + 5*x - 1/2 - sin(4) + 17/4

                     8 x + y - sin(4) + 27/4
```

The ordering of the terms of a sum is not necessarily the same as on input:

```
>> x + y + z + a + b + c
```

```
                    a + b + c + x + y + z
```

```
>> 1 + x + x^2 + x^10
```

```
                           2    10
                      x + x  + x   + 1
```

Internally, this sum is a symbolic call of `_plus`:

```
>> op(%, 0), type(%)
```

```
                       _plus, "_plus"
```

Example 2. The functional equivalent `_plus` of the operator `+` is a handy tool for computing finite sums. In the following, the terms are generated via the sequence operator `$`:

```
>> _plus(i^2 $ i = 1..100)
```

```
                          338350
```

E.g., it is easy to add up all elements in a set:

```
>> S := {a, b, 1, 2, 27}: _plus(op(S))
```

```
                        a + b + 30
```

The following command "zips" two lists by adding corresponding elements:

```
>> L1 := [a, b, c]: L2 := [1, 2, 3]: zip(L1, L2, _plus)
```

```
                   [a + 1, b + 2, c + 3]
```

```
>> delete S, L1, L2:
```

Example 3. Polynomials of type `DOM_POLY` are added by `+`, if they have the same indeterminates and the same coefficient ring:

```
>> poly(x^2 + 1, [x]) + poly(x^2 + x - 1, [x])
```

```
                              2
                     poly(2 x  + x, [x])
```

Symbolic sums are returned if the indeterminates or the coefficient rings do not match:

```
>> poly(x, [x]) + poly(x, [x, y])
```

```
               poly(x, [x]) + poly(x, [x, y])
```

```
>> poly(x, [x]) + poly(x, [x], Dom::Integer)
```

```
         poly(x, [x]) + poly(x, [x], Dom::Integer)
```

Example 4. Adding a constant to an equality, an inequality, or a comparison amounts to adding it to both sides:

```
>> (a = b) + c, (a <> b) + c, (a <= b) + c, (a < b) + c

  a + c = b + c, a + c <> b + c, a + c <= b + c, a + c < b + c
```

Adding an equality is performed by adding the left hand sides and the right hand sides separately:

```
>> (a = b) + (c = d), (a <> b) + (c = d),
   (a <= b) + (c = d), (a < b) + (c = d)

  a + c = b + d, a + c <> b + d, a + c <= b + d, a + c < b + d
```

Inequalities can only be added to equalities:

```
>> (a = b) + (c <> d), (a <> b) + (c <> d),
   (a <= b) + (c <> d), (a < b) + (c <> d)

               a + c <> b + d, FAIL, FAIL, FAIL
```

The addition of comparisons takes of the difference between $<$ and $\leq$ into account. Note that MuPAD uses only these two comparison operators; $>$ and $\geq$ are automatically rewritten:

```
>> (a = b) + (c <= d), (a <> b) + (c <= d),
   (a <= b) + (c <= d), (a < b) + (c <= d);

       a + c <= b + d, FAIL, a + c <= b + d, a + c < b + d

>> (a = b) + (c < d), (a <> b) + (c < d),
   (a <= b) + (c < d), (a < b) + (c < d);

       a + c < b + d, FAIL, a + c <= b + d, a + c < b + d

>> (a = b) + (c >= d), (a <> b) + (c >= d),
   (a <= b) + (c >= d), (a < b) + (c >= d);

       a + d <= b + c, FAIL, a + d <= b + c, a + d < b + c

>> (a = b) + (c > d), (a <> b) + (c > d),
   (a <= b) + (c > d), (a < b) + (c > d);

       a + d < b + c, FAIL, a + d <= b + c, a + d < b + c
```

Example 5. For finite sets `X`, `Y`, the sum `X + Y` is the set $\{x+y;\ x \in X,\ y \in Y\}$:

```
>> {a, b, c} + {1, 2}

           {a + 1, a + 2, b + 1, b + 2, c + 1, c + 2}
```

Example 6. Various library domains such as matrix domains overload `_plus`:

```
>> x := Dom::Matrix(Dom::Integer)([1, 2]):
   y := Dom::Matrix(Dom::Rational)([2, 3]):
   x + y, y + x
```

```
                        +-   -+  +-   -+
                        |  3  |  |  3  |
                        |     |, |     |
                        |  5  |  |  5  |
                        +-   -+  +-   -+
```

If the terms in a sum `x + y` are of different type, the first term `x` tries to convert `y` to the data type of `x`. If successful, the sum is of the same type as `x`. In the previous example, `x` and `y` have different types (both are matrices, but the component domains differ). Hence the sums `x + y` and `y + x` differ syntactically, because they inherit their type from the first term:

```
>> bool(x + y = y + x)

                              FALSE
```

```
>> domtype(x + y), domtype(y + x)

      Dom::Matrix(Dom::Integer), Dom::Matrix(Dom::Rational)
```

If `x` does not succeed to convert `y`, then `FAIL` is returned. In the following call, the component `2/3` cannot be converted to an integer:

```
>> y := Dom::Matrix(Dom::Rational)([2/3, 3]): x + y

                              FAIL
```

```
>> delete x, y:
```

Example 7. This example demonstrates how to implement a slot `"_plus"` for a domain. The following domain `myString` is to represent character strings. The sum of such strings is to be the concatenation of the strings.

The `"new"` method uses `expr2text` to convert any MuPAD object to a string. This string is the internal representation of elements of `myString`. The `"print"` method turns this string into the screen output:

```
>> myString := newDomain("myString"):
   myString::new := proc(x)
   begin
     if args(0) = 0 then x := "": end_if;
     case domtype(x)
       of myString do return(x);
```

```
      of DOM_STRING do return(new(dom, x));
      otherwise return(new(dom, expr2text(x)));
    end_case
  end_proc:
  myString::print := x -> extop(x, 1):
```

Without a "_plus" method, the system function _plus handles elements of this domain like any symbolic object:

```
>> y := myString(y): z := myString(z): 1 + x + y + z + 3/2

                          x + y + z + 5/2
```

Now, we implement the "_plus" method. It checks all arguments. Arguments are converted, if they are not of type myString. Generally, such an implicit conversion should be avoided. In this case, however, any object has a corresponding string representation via expr2text and an implicit conversion is implemented. Finally, the sum of myString objects is defined as the concatenation of the internal strings:

```
>> myString::_plus := proc()
   local n, Arguments, i;
   begin
     userinfo(10, "myString::_plus called with the arguments:",
              args()):
     n := args(0):
     Arguments := [args()];
     for i from 1 to n do
       if domtype(Arguments[i]) <> myString then
         // convert the i-th term to myString
         Arguments[i] := myString::new(Arguments[i]):
       end_if;
     end_for:
     myString::new(_concat(extop(Arguments[i], 1) $ i = 1..n))
   end_proc:

   setuserinfo(myString::_plus, 10):
```

Now, myString objects can be added:

```
>> myString("This ") + myString("is ") + myString("a string")

 Info: myString::_plus called with the arguments:, This , is , \
 a string

                         This is a string
```

In the following sum, y and z are elements of myString. The term y is the first term that is an element of a library domain. Its "_plus" method is called and concatenates all terms to a string of type myString:

```
>> 1 + x + y + z + 3/2;

 Info: myString::_plus called with the arguments:, 1, x, y, z, \
 3/2

                          1xyz3/2

>> delete myString, y, z:
```

Changes:

- Addition of equalities, inequalities, and comparisons was added.

- – subtract expressions

`x - y` computes the difference of `x` and `y`.

Call(s):

- `x - y`
- `_subtract(x, y)`

Parameters:

`x, y` — arithmetical expressions, polynomials of type `DOM_POLY`, or sets

Return Value: an arithmetical expression, a polynomial, or a set.

Overloadable by: `x, y`

Related Functions: `_invert`, `_negate`, `^`, `/`, `*`, `+`, `poly`, `Pref::keepOrder`

Details:

- `x - y` is equivalent to the function call `_subtract(x, y)`.
- For numbers of type `Type::Numeric`, the difference is returned as a number.
- If neither `x` nor `y` are elements of library domains with `"_subtract"` methods, `x - y` is internally represented as `x + y*(-1)` = `_plus(x, _mult(y, -1))`.

- If `x` or `y` is an element of a domain with a slot `"_subtract"`, then this method is used to compute `x - y`. Many library domains overload the `-` operator by an appropriate `"_subtract"` slot. Differences are processed as follows:

  `x - y` is searched for elements of library domains from left to right. Let `z` (either `x` or `y`) be the first term that is not of one of the basic types provided by the kernel (numbers, expressions, etc.). If the domain `d` = `z::dom` = `domtype(z)` has a slot `"_subtract"`, it is called in the form `d::_subtract(x, y)`. The result returned by `d::_subtract` is the result of `x - y`.

  Users should implement the slot `d::_subtract` of their domains `d` according to the following convention:

  - If both `x` and `y` are elements of `d`, an appropriate difference of type `d` should be returned.
  - If either `x` or `y` cannot be converted to an element of `d`, the slot should return `FAIL`.
  - Care must be taken if either `x` or `y` is not of type `d`, but can be converted to type `d`. This object should be converted only if the mathematical semantics is obvious to any user who uses this domain as a 'black box' (e.g., integers may be regarded as rational numbers because of the natural mathematical embedding). If in doubt, the `"_subtract"` method should return `FAIL` instead of using implicit conversions. If implicit conversions are used, they must be well-documented.

  Cf. examples 4 and 5.

  Most of the library domains in MuPAD's standard installation comply with this convention.

- Polynomials of type `DOM_POLY` are subtracted by `-`, if they have the same indeterminates and the same coefficient ring.
- For finite sets `X`, `Y`, the difference `X - Y` is the set $\{x - y;\ x \in X,\ y \in Y\}$.
- `_subtract` is a function of the system kernel.

Example 1. The difference of numbers is simplified to a number:

```
>> 1234 - 234, I + x - y - 4*I, 3 + x - y - 29/3

                   1000, x - y - 3 I, x - y - 20/3
```

Internally, a symbolic difference `x - y` is represented as the sum `x + y*(-1)`:

```
>> type(x - y), op(x - y, 0), op(x - y, 1), op(x - y, 2)
```

```
                    "_plus", _plus, x, -y
```

```
>> op(op(x - y, 2))

                              y, -1
```

Example 2. Polynomials of type DOM_POLY are subtracted by -, if they have the same indeterminates and the same coefficient ring:

```
>> poly(x^2 + 1, [x]) - poly(x^2 + x - 1, [x])

                        poly(- x + 2, [x])
```

Symbolic differences are returned if the indeterminates or the coefficient rings do not match:

```
>> poly(x, [x]) - poly(x, [x, y])

               poly(x, [x]) + poly((-1) x, [x, y])

>> poly(x, [x]) - poly(x, [x], Dom::Integer)

          poly(x, [x]) + poly((-1) x, [x], Dom::Integer)
```

Example 3. For finite sets X, Y, the difference X - Y is the set $\{x - y;\ x \in X,\ y \in Y\}$:

```
>> {a, b, c} - {1, 2}

           {a - 1, a - 2, b - 1, b - 2, c - 1, c - 2}
```

Example 4. Various library domains such as matrix domains overload _subtract:

```
>> x := Dom::Matrix(Dom::Integer)([2, 2]):
   y := Dom::Matrix(Dom::Rational)([1, 3]):
   x - y, y - x

                        +-     -+  +-     -+
                        |   1   |  |  -1   |
                        |       |, |       |
                        |  -1   |  |   1   |
                        +-     -+  +-     -+
```

If the terms in x - y are of different type, the first term x tries to convert y to the data type of x. If successful, the difference is of the same type as x. In the previous example, x and y have different types (both are matrices, but the component domains differ). Consequently, x - y and y - x have different types, because they inherit their type from the first term:

```
>> domtype(x - y), domtype(y - x)

      Dom::Matrix(Dom::Integer), Dom::Matrix(Dom::Rational)
```

If x does not succeed to convert y, then FAIL is returned. In the following call, the component 2/3 cannot be converted to an integer:

```
>> y := Dom::Matrix(Dom::Rational)([2/3, 3]): x - y

                                FAIL
```

The matrix domain defines x - y as x + (-y):

```
>> x::dom::_subtract

            (x, y) -> dom::_plus(x, dom::_negate(y))

>> delete x, y:
```

Example 5. This example demonstrates how to implement a slot "_subtract" for a domain. The following domain myString is to represent character strings. The difference x - y of such strings is to remove all characters in y from x.

The "new" method uses expr2text to convert any MuPAD object to a string. This string is the internal representation of elements of myString. The "print" method turns this string into the screen output:

```
>> myString := newDomain("myString"):
   myString::new := proc(x)
   begin
     if args(0) = 0 then x := "" end_if;
     case domtype(x)
       of myString do return(x);
       of DOM_STRING do return(new(dom, x));
       otherwise return(new(dom, expr2text(x)));
     end_case
   end_proc:
   myString::print := x -> extop(x, 1):
```

Without a "_subtract" method, the system handles elements of this domain like any symbolic object:

```
>> x := myString(x): y := myString(y): x - y
```

```
x - y
```

Now, we implement the "_subtract" method. It checks all arguments. Arguments are converted if they are not of type myString. Generally, such an implicit conversion should be avoided. In this case, however, any object has a corresponding string representation via expr2text and an implicit conversion is implemented. Finally, the difference x - y of myString objects removes all characters in the string y from the string x:

```
>> myString::_subtract := proc(x, y)
   local i, char;
   begin
     userinfo(10, "myString::_subtract called with ".
              "the arguments:", args()):
     // Convert all arguments to myString.
     if domtype(x) <> myString then x := myString::new(x) end_if;
     if domtype(y) <> myString then y := myString::new(y) end_if;
     // extract the internal strings
     x := extop(x, 1):
     y := extop(y, 1):
     // convert the strings to a list/set of characters
     x := [x[i] $ i = 0 .. length(x) - 1];
     y := {y[i] $ i = 0 .. length(y) - 1};
     // remove all characters in y from x
     for char in y do
       x := subs(x, char = null());
     end_for:
     // concat the remaining characters in x
     myString::new(_concat(op(x)))
   end_proc:

   setuserinfo(myString::_subtract, 10):
```

Now, myString objects can be subtracted:

```
>> myString("This is a string") - myString("is")

 Info: myString::_subtract called with the arguments:, This is \
 a string, is

                              Th  a trng
```

In the following, y is the first term that is an element of a library domain with a "_subtract" slot. This slot is called, converts xyz to an element of myString, and removes the character y:

```
>> xyz - y
```

```
 Info: myString::_subtract called with the arguments:, xyz, y

                                   xz
```

The following `xyz - x - y` = `(xyz - x) - y` calls the `"_subtract"` method twice:

```
>> xyz - x - y

 Info: myString::_subtract called with the arguments:, xyz, x
 Info: myString::_subtract called with the arguments:, yz, y

                                    z

>> delete myString, x, y:
```

---

* – multiply expressions

`x * y * ...` computes the product of `x`, `y` etc.

Call(s):

- `x * y * ...`
- `_mult(x, y, ...)`

Parameters:

`x, y, ...` — arithmetical expressions, polynomials of type `DOM_POLY`, sets, equations, inequalities, or comparisons

Return Value: an arithmetical expression, a polynomial, a set, an equation, an inequality, or a comparison.

Overloadable by: `x, y, ...`

Related Functions: `^`, `/`, `+`, `-`, `_invert`, `_negate`, `poly`, `Pref::timesDot`, `product`

Details:

- `x * y * ...` is equivalent to the function call `_mult(x, y, ...)`.
- All terms that are numbers of type `Type::Numeric` are automatically combined to a single number.

- The terms of a symbolic product may be rearranged internally if no term belongs to a library domain that overloads `_mult`: on terms composed of kernel domains (numbers, identifiers, expressions etc.), multiplication is assumed to be commutative. Cf. example 1.

  Via overloading, the user can implement a non-commutative product for special domains.

- `_mult` accepts an arbitrary number of arguments. In conjunction with the sequence operator `$`, this function is the recommended tool for computing finite products. Cf. example 2. The function `product` may also serve for computing such products. However, `product` is designed for the computation of symbolic and infinite products. It is slower than `_mult`.

- The quotient `x/y` is internally represented as `x * (1/y) = _mult(x, _power(y, -1))`. See `_divide` for details.

- Many library domains overload `_mult` by an appropriate slot `"_mult"`. Products involving elements of library domains are processed as follows:

  A product `x * y * ...` is searched for elements of library domains from left to right. Let `z` be the first term that is not of one of the basic types provided by the kernel (numbers, expressions, etc.). If the domain `d = z::dom = domtype(z)` has a slot `"_mult"`, it is called in the form `d::_mult(x, y, ...)`. The result returned by `d::_mult` is the result of `x * y * ...`.

  Cf. examples 6 and 7.

- `_mult()` returns the number 1.

- Polynomials of type `DOM_POLY` are multiplied by `*`, if they have the same indeterminates and the same coefficient ring. Use `multcoeffs` to multiply polynomials with scalar factors.

- For finite sets `X`, `Y`, the product `X * Y` is the set $\{x\, y;\ x \in X,\ y \in X\}$.

- Equalities, inequalities, and comparisons can be multiplied with one another or with arithmetical expressions. The results of such combinations are demonstrated in example 5.

- `_mult` is a function of the system kernel.

Example 1. Numerical terms are simplified automatically:

```
>> 3 * x * y * (1/18) * sin(4) * 4

                        2 x y sin(4)
                        ------------
                             3
```

The ordering of the terms of a product is not necessarily the same as on input:

```
>> x * y * 3 * z * a * b * c
```

```
3 a b c x y z
```

Internally, this product is a symbolic call of `_mult`:

```
>> op(%, 0), type(%)
```

```
_mult, "_mult"
```

Note that the screen output does not necessarily reflect the internal order of the terms in a product:

```
>> op(%2)
```

```
a, b, c, x, y, z, 3
```

In particular, a numerical factor is internally stored as the last operand. On the screen, a numerical factor is displayed in front of the remaining terms:

```
>> 3 * x * y * 4
```

```
12 x y
```

```
>> op(%)
```

```
x, y, 12
```

Example 2. The functional equivalent `_mult` of the operator `*` is a handy tool for computing finite products. In the following, the terms are generated via the sequence operator `$`:

```
>> _mult(i $ i = 1..20)
```

```
2432902008176640000
```

E.g., it is easy to multiply all elements in a set:

```
>> S := {a, b, 1, 2, 27}: _mult(op(S))
```

```
54 a b
```

The following command "zips" two lists by multiplying corresponding elements:

```
>> L1 := [1, 2, 3]: L2 := [a, b, c]: zip(L1, L2, _mult)
```

```
[a, 2 b, 3 c]
```

```
>> delete S, L1, L2:
```

Example 3. Polynomials of type `DOM_POLY` are multiplied by `*`, if they have the same indeterminates and the same coefficient ring:

```
>> poly(x^2 + 1, [x]) * poly(x^2 + x - 1, [x])

                          4    3
                  poly(x  + x  + x - 1, [x])
```

Symbolic products are returned if the indeterminates or the coefficient rings do not match:

```
>> poly(x, [x]) * poly(x, [x, y])

                 poly(x, [x]) poly(x, [x, y])

>> poly(x, [x]) * poly(x, [x], Dom::Integer)

            poly(x, [x]) poly(x, [x], Dom::Integer)
```

Multiplication of polynomials with scalar factors cannot be achieved with `*`:

```
>> 2 * y * poly(x, [x])

                       2 poly(x, [x]) y
```

Use `multcoeffs` instead:

```
>> multcoeffs(poly(x^2 - 2, [x]), 2*y)

                              2
                   poly((2 y) x  - 4 y, [x])
```

Example 4. For finite sets `X`, `Y`, the product `X * Y` is the set $\{x\,y;\ x \in X,\ y \in Y\}$:

```
>> {a, b, c} * {1, 2}

                  {a, b, c, 2 a, 2 b, 2 c}
```

Note that complex numbers of type `DOM_INT`, `DOM_RAT`, `DOM_COMPLEX`, and `DOM_FLOAT` are implicitly converted to one-element sets, while identifiers are not:

```
>> 2 * {a, b, c}

                       {2 a, 2 b, 2 c}

>> a * {b, c}, PI * {3, 4}

                    {b, c} a, {3, 4} PI
```

Example 5. Multiplying by a constant expression is performed on both sides of an equation:

```
>> (a = b) * c
```

```
                    a c = b c
```

For inequalities, this step is only performed if the constant is known to be nonzero:

```
>> assume(d <> 0):
   (a <> b) * c, (a <> b) * d;
   delete d:
```

```
              (a <> b) c, a d <> b d
```

The multiplication of a comparison with a constant is only defined for real numbers. Even for these, the result depends on the sign of the constant, since multiplication with a negative constant changes the direction of the comparison:

```
>> (a < b) * 2, (a < b) * (-3)
```

```
              2 a < 2 b, -3 b < -3 a
```

```
>> (a < b) * I
```

```
 Error: Inequalities must not be multiplied by complex numbers \
 [_less::_mult]
```

```
>> (a < b) * c, (a <= b) * c
```

```
 piecewise(a c < b c if 0 < c, b c < a c if c < 0),

    piecewise(a c <= b c if 0 <= c, b c <= a c if c <= 0)
```

Multiplication of two equalities is performed by multiplying the left hand sides and the right hand sides separately:

```
>> (a = b) * (c = d)
```

```
                    a c = b d
```

Inequalities cannot be multiplied with one another or with comparisons; multiplication with equalities is, however, defined, if at least one operand of the equation is known to be nonzero:

```
>> assume(d <> 0):
   (a <> b) * (c = d);
   delete d:
```

```
                    a c <> b d
```

In other cases, the product is not expanded:

```
>> delete c, d:
   (a <> b) * (c = d)

                 (a <> b) (c = d)
```

Multiplication of comparisons with equalities and comparisons is performed similar to the cases above:

```
>> assume(c > 0):
   (a < b) * (c = d);
   delete c:

                    a c < b d

>> (a <= b) * (c <= d)

 piecewise(a c <= b d if 0 <= c and 0 <= d,

    b d <= a c if c <= 0 and d <= 0)
```

Example 6. Various library domains such as matrix domains overload `_mult`. The multiplication is not commutative:

```
>> x := Dom::Matrix(Dom::Integer)([[1, 2], [3, 4]]):
   y := Dom::Matrix(Dom::Rational)([[10, 11], [12, 13]]):
   x * y, y * x

               +-        -+  +-        -+
               |  34, 37  |  |  43, 64  |
               |          |, |          |
               |  78, 85  |  |  51, 76  |
               +-        -+  +-        -+
```

If the terms in `x * y` are of different type, the first term `x` tries to convert `y` to the data type of `x`. If successful, the product is of the same type as `x`. In the previous example, `x` and `y` have different types (both are matrices, but the component domains differ). Hence `x * y` and `y * x` have different types that is inherited from the first term:

```
>> domtype(x * y), domtype(y * x)

      Dom::Matrix(Dom::Integer), Dom::Matrix(Dom::Rational)
```

If x does not succeed to convert y, then y tries to convert x. In the following call, the component 27/2 cannot be converted to an integer. Consequently, in x * y, the term y converts x and produces a result that coincides with the domain type of y:

```
>> y := Dom::Matrix(Dom::Rational)([[10, 11], [12, 27/2]]):
   x * y, y * x

                 +-         -+  +-            -+
                 |  34, 38  |  |    43,  64  |
                 |          |, |              |
                 |  78, 87  |  |  105/2, 78  |
                 +-         -+  +-            -+
```

```
>> domtype(x * y), domtype(y * x)

      Dom::Matrix(Dom::Rational), Dom::Matrix(Dom::Rational)
```

```
>> delete x, y:
```

Example 7. This example demonstrates how to implement a slot "_mult" for a domain. The following domain myString is to represent character strings. Via overloading of _mult, integer multiples of such strings should produce the concatenation of an appropriate number of copies of the string.

The "new" method uses expr2text to convert any MuPAD object to a string. This string is the internal representation of elements of myString. The "print" method turns this string into the screen output:

```
>> myString := newDomain("myString"):
   myString::new := proc(x)
   begin
     if args(0) = 0 then x := "": end_if;
     case domtype(x)
       of myString do return(x);
       of DOM_STRING do return(new(dom, x));
       otherwise return(new(dom, expr2text(x)));
     end_case
   end_proc:
   myString::print := x -> extop(x, 1):
```

Without a "_mult" method, the system function _mult handles elements of this domain like any symbolic object:

```
>> y := myString(y): z := myString(z): 4 * x * y * z * 3/2

                           6 x y z
```

Now, we implement the "_mult" method. It uses split to pick out all integer terms in its argument list and multiplies them. The result is an integer n. If there is exactly one other term left (this must be a string of type myString), it is copied n times. The concatenation of the copies is returned:

```
>> myString::_mult:= proc()
   local Arguments, intfactors, others, dummy, n;
   begin
     userinfo(10, "myString::_mult called with the arguments:",
              args());
     Arguments := [args()];
     // split the argument list into integers and other factors:
     [intfactors, others, dummy] :=
         split(Arguments, testtype, DOM_INT);
     // multiply all integer factors:
     n := _mult(op(intfactors));
     if nops(others) <> 1 then
        return(FAIL)
     end_if;
     myString::new(_concat(extop(others[1], 1) $ n))
   end_proc:

   setuserinfo(myString::_mult, 10):
```

Now, integer multiples of myString objects can be constructed via the * operator:

```
>> 2 * myString("string") * 3

 Info: myString::_mult called with the arguments:, 2, string, 3

              stringstringstringstringstringstring
```

Only products of integers and myString objects are allowed:

```
>> 3/2 * myString("a ") * myString("string")

 Info: myString::_mult called with the arguments:, 3/2, a , str\
 ing

                                FAIL

>> delete myString, y, z:
```

Changes:

- Multiplication of sets with identifiers and expressions has changed.

- Multiplication of equalities, inequalities, and comparisons was implemented.

/ – divide expressions

`x/y` computes the quotient of `x` and `y`.

Call(s):

- `x/y`
- `_divide(x, y)`

Parameters:

`x, y, ...` — arithmetical expressions, polynomials of type `DOM_POLY`, or sets

Return Value: an arithmetical expression, a polynomial, or a set.

Overloadable by: `x, y`

Related Functions: `_invert`, `_negate`, `^`, `*`, `+`, `-`, `div`, `divide`, `pdivide`, `poly`

Details:

- `x/y` is equivalent to the function call `_divide(x, y)`.
- For numbers of type `Type::Numeric`, the quotient is returned as a number.
- If neither `x` nor `y` are elements of library domains with `"_divide"` methods, `x/y` is internally represented as `x * y^(-1)` = `_mult(x, _power(y, -1))`.
- If `x` or `y` is an element of a domain with a slot `"_divide"`, then this method is used to compute `x/y`. Many library domains overload the `/` operator by an appropriate `"_divide"` slot. Quotients are processed as follows:

  `x/y` is searched for elements of library domains from left to right. Let `z` (either `x` or `y`) be the first term that is not of one of the basic types provided by the kernel (numbers, expressions, etc.). If the domain `d` = `z::dom` = `domtype(z)` has a slot `"_divide"`, it is called in the form `d::_divide(x, y)`. The result returned by `d::_divide` is the result of `x/y`.

  Cf. examples 4 and 5.

- Polynomials of type DOM_POLY can be divided by /, if they have the same indeterminates and the same coefficient ring, and if exact division is possible. The function divide can be used to compute the quotient of polynomials with a remainder term.
- For finite sets X, Y, the quotient X/Y is the set $\{x/y;\ x \in X,\ y \in Y\}$.
- _divide is a function of the system kernel.

Example 1. The quotient of numbers is simplified to a number:

```
>> 1234/234, 7.5/7, 6*I/2

                617/117, 1.071428571, 3 I
```

Internally, a symbolic quotient x/y is represented as the product x * y^(-1):

```
>> type(x/y), op(x/y, 0), op(x/y, 1), op(x/y, 2)

                                   1
                 "_mult", _mult, x, -
                                   y

>> op(op(x/y, 2), 0), op(op(x/y, 2), 1), op(op(x/y, 2), 2)

                       _power, y, -1
```

Example 2. For finite sets X, Y, the quotient X/Y is the set $\{x/y;\ x \in X,\ y \in Y\}$:

```
>> {a, b, c} / {2, 3}

                 { a  a  b  b  c  c }
                 { -, -, -, -, -, - }
                 { 2  3  2  3  2  3 }
```

Example 3. Polynomials of type DOM_POLY can be divided by / if they have the same indeterminates, the same coefficient ring, and if exact division is possible:

```
>> poly(x^2 - 1, [x]) / poly(x - 1, [x])

                   poly(x + 1, [x])

>> poly(x^2 - 1, [x]) / poly(x - 2, [x])

                        FAIL
```

The function `divide` provides division with a remainder:

```
>> divide(poly(x^2 - 1, [x]), poly(x - 2, [x]))

                 poly(x + 2, [x]), poly(3, [x])
```

The polynomials must have the same indeterminates and the same coefficient ring:

```
>> poly(x^2 - 1, [x, y]) / poly(x - 1, [x])

 Error: Illegal argument [divide]
```

Example 4. Various library domains such as matrix domains overload `_divide`. The matrix domain defines `x/y` as `x * (1/y)`, where `1/y` is the inverse of `y`:

```
>> x := Dom::Matrix(Dom::Integer)([[1, 2], [3, 4]]):
   y := Dom::Matrix(Dom::Rational)([[10, 11], [12, 13]]):
   x/y

                        +-            -+
                        |  11/2, -9/2  |
                        |              |
                        |   9/2, -7/2  |
                        +-            -+
```

The inverse of `x` has rational entries. Therefore, `1/x` returns `FAIL`, because the component ring of `x` is `Dom::Integer`. Consequently, also `y/x` returns `FAIL`:

```
>> y/x

                              FAIL

>> delete x, y:
```

Example 5. This example demonstrates the behavior of `_divide` on user-defined domains. In the first case below, the user-defined domain does not have a `"_divide"` slot. Thus `x/y` is transformed to `x * (1/y)`:

```
>> Do := newDomain("Do"): x := new(Do, 1): y := new(Do, 2):
   x/y; op(x/y, 0..2)

                           new(Do, 1)
                           ----------
                           new(Do, 2)

                                        1
                 _mult, new(Do, 1), ----------
                                    new(Do, 2)
```

After the slot "_divide" is defined in the domain Do, this method is used to divide elements:

```
>> Do::_divide := proc() begin "The Result" end: x/y

                              "The Result"

>> delete Do, x, y:
```

---

^ – raise an expression to a power

x^y computes the y-th power of x.

Call(s):

- x^y
- _power(x, y)

Parameters:

x, y — arithmetical expressions, polynomials of type DOM_POLY, floating point intervals, or sets

Return Value: an arithmetical expression, a polynomial, a floating point interval, or a set.

Overloadable by: x, y

Related Functions: _invert, _negate, *, /, +, -, numlib::ispower, powermod

---

Details:

- x^y is equivalent to the function call _power(x, y).
- The power operator ^ is left associative: x^y^z is parsed as (x^y)^z. Cf. example 2.
- If x is a polynomial of type DOM_POLY, then y must be a nonnegative integer smaller than 2^{31}.
- _power is overloaded for matrix domains (matrix). In particular, x^(-1) returns the inverse of the matrix x.
- Use powermod to compute modular powers. Cf. example 3.

- Mathematically, the call `sqrt(x)` is equivalent to `x^(1/2)`. Note, however, that `sqrt` tries to simplify the result. Cf. example 4.
- If `x` or `y` is an element of a domain with a slot `"_power"`, then this method is used to compute `x^y`. Many library domains overload the `^` operator by an appropriate `"_power"` slot. Powers are processed as follows:

  `x^y` is searched for elements of library domains from left to right. Let `z` (either `x` or `y`) be the first term that is not of one of the basic types provided by the kernel (numbers, expressions, etc.). If the domain `d` = `z::dom` = `domtype(z)` has a slot `"_power"`, it is called in the form `d::_power(x, y)`. The result returned by `d::_power` is the result of `x^y`.

  Cf. examples 6 and 7.
- For finite sets `X`, `Y`, the power `X^Y` is the set $\{x^y;\ x \in X,\ y \in Y\}$.
- `_power` is a function of the system kernel.

Example 1. Some powers are computed:

```
>> 2^10, I^(-5), 0.3^(1/3), x^(1/2) + y^(-1/2), (x^(-10) + 1)^2

                                 1/2      1    /  1       \2
        1024, -I, 0.6694329501, x    + ----, | --- + 1 |
                                        1/2   |  10      |
                                       y      \ x        /
```

Use `expand` to “expand” powers of sums:

```
>> (x + y)^2 = expand((x + y)^2)

                          2             2    2
                   (x + y)  = 2 x y + x  + y
```

Note that identities such as `(x*y)^z` = `x^z * y^z` only hold in certain areas of the complex plane:

```
>> ((-1)*(-1))^(1/2) <> (-1)^(1/2) * (-1)^(1/2)

                              1 <> -1
```

Consequently, the following `expand` command does not expand its argument:

```
>> expand((x*y)^(1/2))

                                 1/2
                            (x y)
```

Example 2. The power operator ^ is left associative:

```
>> 2^3^4 = (2^3)^4, x^y^z

                                    y z
                       4096 = 4096, (x )
```

Example 3. Modular powers can be computed directly using ^ and `mod`. However, `powermod` is more efficient:

```
>> 123^12345 mod 17 = powermod(123, 12345, 17)

                              4 = 4
```

Example 4. The function `sqrt` produces simpler results than `_power`:

```
>> sqrt(4*x*y), (4*x*y)^(1/2)

                           1/2          1/2
                    2 (x y)   , (4 x y)
```

Example 5. For finite sets, `X^Y` is the set $\{x^y;\ x \in X,\ y \in Y\}$:

```
>> {a, b, c}^2, {a, b, c}^{q, r, s}

         2   2   2     q   r   q   s   r   q   s   r   s
       {a , b , c }, {a , a , b , a , b , c , b , c , c }
```

Example 6. Various library domains such as matrix domains or residue class domains overload `_power`:

```
>> x := Dom::Matrix(Dom::IntegerMod(7))([[2, 3], [3, 4]]):
   x^2, x^(-1), x^3 * x^(-3)

 +-                  -+  +-                  -+
 |  6 mod 7, 4 mod 7  |  |  3 mod 7, 3 mod 7  |
 |                    |, |                    |,
 |  4 mod 7, 4 mod 7  |  |  3 mod 7, 5 mod 7  |
 +-                  -+  +-                  -+

    +-                  -+
    |  1 mod 7, 0 mod 7  |
    |                    |
    |  0 mod 7, 1 mod 7  |
    +-                  -+
```

```
>> delete x:
```

Example 7. This example demonstrates the behavior of `_power` on user-defined domains. Without a `"power"` slot, powers of domain elements are handled like any other symbolic powers:

```
>> myDomain := newDomain("myDomain"): x := new(myDomain, 1): x^2

                                     2
                  (new(myDomain, 1))
```

```
>> type(x^2), op(x^2, 0), op(x^2, 1), op(x^2, 2)

            "_power", _power, new(myDomain, 1), 2
```

After the `"_power"` slot is defined, this method is used to compute powers of `myDomain` objects:

```
>> myDomain::_power := proc() begin "The result" end: x^2

                        "The result"
```

```
>> delete myDomain, x:
```

Changes:

- Floating point intervals are handled.

`@` – compose functions

`f@g` represents the composition $x \mapsto f(g(x))$ of the functions `f` and `g`.

Call(s):

- `f @ g @ ...`
- `_fconcat(f, g, ...)`

Parameters:

`f, g, ...` — functions

Return Value: an expression of type `"_fconcat"`.

Overloadable by: `f, g, ...`

Related Functions: `@@`

Details:

- In MuPAD, functions are usually represented by procedures of type `DOM_PROC`, function environments, or functional expressions such as `f@g@exp + id^2`. In fact, practically any MuPAD object may serve as a function.
- `f @ g` is equivalent to the function call `_fconcat(f, g)`.
- `_fconcat()` returns the identity map `id`; `_fconcat(f)` returns `f`.
- `_fconcat` is a function of the system kernel.

Example 1. The following function `h` is the composition of the system functions `abs` and `sin`:

```
>> h := abs@sin

                            abs@sin

>> h(x), h(y + 2), h(0.5)

           abs(sin(x)), abs(sin(y + 2)), 0.4794255386
```

The following functional expressions represent polynomials:

```
>> f := id^3 + 3*id - 1: f(x), (f@f)(x)

                3                3           3     3
         3 x + x  - 1, 9 x + 3 x  + (3 x + x  - 1)  - 4
```

The random generator `random` produces nonnegative integers with 12 digits. The following composition of `float` and `random` produces random floating point numbers between 0.0 and 1.0:

```
>> rand := float@random/10^12: rand() $ k = 1..12

 0.4274196691, 0.3211106933, 0.3436330737, 0.4742561436,

    0.558458719, 0.7467538305, 0.03206222208, 0.7229741218,

    0.6043056139, 0.7455800374, 0.2598119527, 0.3100754872
```

In conjunction with the function `map`, the composition operator `@` is a handy tool to apply composed functions to the operands of a data structure:

```
>> map([1, 2, 3, 4], (PI + id^2)@sin),
   map({1, 2, 3, 4}, cos@float)
```

```
               2               2              2               2
 [PI + sin(1) , PI + sin(2) , PI + sin(3) , PI + sin(4) ],

    {-0.9899924966, -0.6536436209, -0.4161468365, 0.5403023059}
```

```
>> delete h, f, rand:
```

Example 2. Some simplifications of functional expressions are possible via `simplify`:

```
>> cos@arccos + exp@ln = simplify(cos@arccos + exp@ln)

                  cos@arccos + exp@ln = 2 id
```

`@@` – iterate a function

`f@@n` represents the `n`-fold iterate `x -> f(f(...(f(x))...))` of the function `f`.

Call(s):

- `f @@ n`
- `_fnest(f, n)`

Parameters:

`f` — a function
`n` — an integer

Return Value: a function

Related Functions: `@`, `fp::fixargs`, `fp::nest`, `fp::nestvals`, `fp::fold`

Details:

- The statement `f@@n` is equivalent to the call `_fnest(f, n)`.
- For positive `n`, `f@@n` is also equivalent to `_fconcat(f $ n)`.
- `f@@0` returns the identity map `id`.
- If `f` is a function environment with the slot `"inverse"` set, `n` can also be negative. Cf. example 2.

- Iteration is only reasonable for functions that accept their own return values as input. Note that `fp::fixargs` is a handy tool for converting functions with parameters to univariate functions which may be suitable for iteration. Cf. example 3.

Example 1. For a nonnegative integer `n`, `f@@n` is equivalent to an `_fconcat` call:

```
>> f@@4, (f@@4)(x)
```

```
f@f@f@f, f(f(f(f(x))))
```

`@@` simplifies the composition of symbolic iterates:

```
>> (f@@n)@@m
```

```
f@@(m n)
```

The iterate may be called like any other MuPAD function. If `f` evaluates to a procedure and `n` to an integer, a corresponding value is computed:

```
>> f := x -> x^2:  (f@@n)(x) $ n = 0..10
```

$$x, x^{2}, x^{4}, x^{8}, x^{16}, x^{32}, x^{64}, x^{128}, x^{256}, x^{512}, x^{1024}$$

```
>> delete f:
```

Example 2. For functions with a known inverse function, `n` may be negative. The function `f` must have been declared as a function environment with the `"inverse"` slot. Examples of such functions include the trigonometric functions which are implemented as function environments in MuPAD:

```
>> sin::"inverse", sin@@-3, (sin@@(-3))(x)
```

```
"arcsin", arcsin@arcsin@arcsin, arcsin(arcsin(arcsin(x)))
```

Example 3. `@@` can only be used for functions that accept their own output domain as an input, i.e., $f : M \mapsto M$ for some set M. If you want to use `@@` with a function which needs additional parameters, `fp::fixargs` is a handy tool to generate a corresponding univariate function. In the following call, the function `f: x -> g(x, p)` is iterated:

```
>> g := (x, y) -> x^2 + y: f := fp::fixargs(g, 1, p): (f@@4)(x)
```

```
                              2 2 2 2
p + (p + (p + (p + x ) ) )

>> delete g, f:
```

$ – create an expression sequence

`$ a..b` creates the sequence of integers from `a` through `b`.

`f $ n` creates the sequence `f, ... , f` consisting of `n` copies of `f`.

`f(i) $ i = a..b` creates the sequence `f(a), f(a+1), ... , f(b)`.

`f(i) $ i in object` creates the sequence `f(i1), f(i2), ...`, where `i1`, `i2` etc. are the operands of the `object`.

Call(s):

- `$ a..b`
- `_seqgen(a..b)`
- `f $ n`
- `_seqgen(f, n)`
- `f $ i = a..b`
- `_seqgen(f, i, a..b)`
- `f $ i in object`
- `_seqin(f, i, object)`

Parameters:

`f, object`	—	arbitrary MuPAD objects
`n, a, b`	—	integers
`i`	—	an identifier or a local variable (`DOM_VAR`) of a procedure

Return Value: an expression sequence of type `"_exprseq"` or the void object of type `DOM_NULL`.

Overloadable by: `a..b, f, n, i, object`

Related Functions: `_exprseq`, `null`

Details:

- The `$` operator is a most useful tool. It serves for generating sequences of objects. Sequences are used to define sets or lists, and may be passed as arguments to system functions. Cf. example 1.
- `$ a..b` and the equivalent function call `_seqgen(a..b)` produce the sequence of integers `a, a + 1, ... , b`. The void object of type `DOM_NULL` is produced if `a > b`.
- `f $ n` and the equivalent function call `_seqgen(f, n)` produce a sequence of `n` copies of the object `f`. Note that `f` is evaluated only once, before the sequence is created. The empty sequence of type `DOM_NULL` is produced if `n` is not positive.
- `f $ i = a..b` and the equivalent function call `_seqgen(f, i, a..b)` successively substitute `i = a` through `i = b` into `f` and evaluates the results. The following expression sequence is produced:

  `eval(subs(f,i=a))`, `eval(subs(f,i=a+1))`, ... , `eval(subs(f,i=b))`.

  Note that `f` is not evaluated before the substitutions. The void object of type `DOM_NULL` is produced if `a > b`.
- `f $ i in object` and the equivalent function call `_seqin(f, i, object)` successively replace `i` by the operands of the `object`: they substitute `i = op(object, 1)` through `i = op(object, n)` into `f` and evaluate the results ($n =$ `nops(object)` is the number of operands). The following expression sequence is produced:

  `eval(subs(f,i=op(object,1)))`, ... ,
  `eval(subs(f,i=op(object,n)))`.

  Note that `f` is not evaluated before the substitutions. The empty sequence of type `DOM_NULL` is produced if the `object` has no operands.
- The "loop variable" `i` in `f $ i = a..b` and `f $ i in object` may have a value. This value is not changed by using `i` inside a `$` statement.
- `_seqgen` is a function of the system kernel.

Example 1. The following sequence can be passed as arguments to the function `_plus`, which adds up its arguments:

```
>> i^2 $ i = 1..5

                         1, 4, 9, 16, 25
```

```
>> _plus(i^2 $ i = 1..5)
```

```
                                    55
```

The 5-th derivative of the expression `exp(x^2)` is:

```
>> diff(exp(x^2), x $ 5)
```

```
                     2          3      2         5      2
          120 x exp(x ) + 160 x  exp(x ) + 32 x  exp(x )
```

We compute the first derivatives of `sin(x)`:

```
>> diff(sin(x), x $ i) $ i = 0..5
```

```
         sin(x), cos(x), -sin(x), -cos(x), sin(x), cos(x)
```

We use `ithprime` to compute the first 10 prime numbers:

```
>> ithprime(i) $ i = 1..10
```

```
                2, 3, 5, 7, 11, 13, 17, 19, 23, 29
```

We select all primes from the set of integers between 1990 and 2010:

```
>> select({$ 1990..2010}, isprime)
```

```
                     {1993, 1997, 1999, 2003}
```

The 3×3 matrix with entries $A_{ij} = i \cdot j$ is generated:

```
>> n := 3: matrix([[i*j $ j = 1..n] $ i = 1..n])
```

```
                          +-         -+
                          |  1, 2, 3  |
                          |           |
                          |  2, 4, 6  |
                          |           |
                          |  3, 6, 9  |
                          +-         -+
```

```
>> delete n:
```

Example 2. In `f $ n`, the object `f` is evaluated only once. The result is copied `n` times. Consequently, the following call produces copies of one single random number:

```
>> random() $ 3
```

```
            427419669081, 427419669081, 427419669081
```

The following call evaluates `random` for each value of of `i`:

```
>> random() $ i = 1..3
```

```
            321110693270, 343633073697, 474256143563
```

Example 3. In the following call, `i` runs through the list:

```
>> i^2 $ i in [3, 2, 1]
```

```
9, 4, 1
```

Note that the screen output of sets does not necessarily coincide with the internal ordering:

```
>> Set := {1, 2, 3, 4}: Set, [op(Set)]
```

```
{1, 2, 3, 4}, [4, 3, 2, 1]
```

The `$` operator respects the internal ordering:

```
>> i^2 $ i in Set
```

```
16, 9, 4, 1
```

```
>> delete Set:
```

Example 4. Arbitrary objects `f` are allowed in `f $ i = a..b`. In the following call, `f` is an assignment (it has to be enclosed in brackets). The sequence computes a table `f[i] = i!`:

```
>> f[0] := 1: (f[i] := i*f[i - 1]) $ i = 1..4: f
```

```
table(
  4 = 24,
  3 = 6,
  2 = 2,
  1 = 1,
  0 = 1
)
```

```
>> delete f:
```

`_exprseq` – expression sequences

The function call `_exprseq(object1, object2, ...)` is the internal representation of the expression sequence `object1, object2, ...`.

Call(s):

- `object1, object2, ...`
- `_exprseq(object1, object2, ...)`

Parameters:

object1, object2, ... — arbitrary MuPAD objects

Return Value: an expression of type "_exprseq" or the void object of type DOM_NULL.

Related Functions: _stmtseq, null

Details:

- In MuPAD, "sequences" are ordered collections of objects separated by commas. You may think of the comma as an operator that concatenates sequences. Internally, sequences are represented as function calls _exprseq(object1, object2, ...). On the screen, sequences are printed as object1, object2, ...
- _exprseq() and the equivalent call null() yield the void object of type DOM_NULL.
- When evaluating an expression sequence, all void objects of type DOM_NULL are removed from it, automatically.
- The $ operator is a useful tool for generating sequences.
- When a MuPAD function or procedure is called with more than one argument, the parameters are passed as an expression sequence.
- _exprseq is a function of the system kernel.

Example 1. A sequence is generated by "concatenating" objects with commas. The resulting object is of type "_exprseq":

```
>> a, b, sin(x)

                    a, b, sin(x)

>> op(%, 0), type(%)

              _exprseq, "_exprseq"
```

On the screen, _exprseq just returns its argument sequence:

```
>> _exprseq(1, 2, x^2 + 5) = (1, 2, x^2 + 5)

                  2               2
         (1, 2, x  + 5) = (1, 2, x  + 5)
```

Example 2. The object of domain `DOM_NULL` (the "empty sequence") is automatically removed from expression sequences:

```
>> 1, 2, null(), 3

                              1, 2, 3
```

Expression sequences are flattened. The following sequence does not have 2 operands, where the second operand is a sequence. Instead, it is flattened to a sequence with 3 operands:

```
>> x := 1: y := 2, 3: x, y

                              1, 2, 3

>> delete x, y:
```

Example 3. Sequences are used to build sets and lists. Sequences can also be passed to functions that accept several arguments:

```
>> s := 1, 2, 3: {s}, [s], f(s)

          {1, 2, 3}, [1, 2, 3], f(1, 2, 3)

>> delete s:
```

`_index` – indexed access

`x[i]` and `x[i1, i2, ...]` yield the entries of `x` corresponding to the indices `i` and `i1, i2, ...`, respectively.

Call(s):

- `x[i]`
- `_index(x, i)`
- `x[i1, i2, ...]`
- `_index(x, i1, i2, ...)`

Parameters:

`x`	—	an arbitrary MuPAD object. In particular, a "container object": a list, a finite set, an array, a matrix, a table, an expression sequence, or a character string.
`i, i1, i2, ...`	—	indices. For most "containers" `x`, indices must be integers. If `x` is a table, arbitrary MuPAD objects can be used as indices.

Return Value: the entry of `x` corresponding to the index. If `x` is not a list, a set, an array etc., an indexed object of type `"_index"` is returned.

Overloadable by: `x`

Related Functions: `:=`, `_assign`, `array`, `contains`, `DOM_ARRAY`, `DOM_LIST`, `DOM_SET`, `DOM_STRING`, `DOM_TABLE`, `indexval`, `op`, `slot`, `table`, `Type::Indeterminate`

Details:

- `x[i]` and `x[i1, i2, ...]` are equivalent to `_index(x, i)` and `_index(x, i1, i2, ...)`, respectively.
- Any MuPAD object `x` allows an indexed call of the form `x[i]` or `x[i1, i2, ...]`. If `x` is not a "container object" (a list, a set, an array etc.), a symbolic indexed object is returned. In particular, "indexed identifiers" are returned if `x` is an identifier. In this case, indices may be arbitrary MuPAD objects. Cf. example 1.
- For lists, finite sets, and expression sequences, the index `i` is restricted to the integers from `1` through `nops(x)`. For lists and sequences, `x[i] = op(x, i)` holds.
- For finite sets, `x[i]` returns the `i`-th element as printed on the screen. Note, however, that the function `op` refers to the *internal* ordering of the elements: in general, `x[i] <> op(x, i)` for sets. Before screen output and indexed access, the elements of sets are sorted via the slot `DOM_SET::sort`.
- For arrays, appropriate indices `i` or multi-indices `i1, i2, ...` from the index range defined by `array` must be used. If any specified index is an integer outside the admissible range, an error occurs. If any specified index is not an integer (e.g., a symbol `i`), then `x[i]` or `x[i1, i2,...]` is returned symbolically. For one-dimensional arrays `x := array(1..n, [...])`, the entries correspond to the operands: `x[i] = op(x, i)`.
- For matrices, appropriate indices `i` or multi-indices `i1, i2, ...` from the index range defined by `matrix` must be used. Indices outside this range or symbolic indices lead to an error. For a one-dimensional matrix representing a column vector, `x[i] = x[i, 1] = op(x, i)` holds. For a one-dimensional matrix representing a row vector, `x[i] = x[1, i] = op(x, i)` holds.
- For tables, any index may be used. If there is no corresponding entry in the table, `x[i]` or `x[i1, i2, ...]` is returned symbolically.

- For character strings, the index `i` is restricted to the integers from `0` through `length(x) - 1`. Note that the first character of a string carries the index `0`!

- The entry returned by an indexed call is fully evaluated. For arrays and tables, evaluation can be suppressed in indexed calls via `indexval`. Cf. example 5.

- Note, that an indexed assignment such as `x[i] := value` implicitly turns `x` into a table with a single entry, if `x` is not one of the "container" types above.

- `_index` is a function of the system kernel.

Example 1. Indexed identifiers are useful when solving equations in many unknowns:

```
>> n := 4:
   equations := {x[i-1] - 2*x[i] + x[i+1] = 1 $ i = 1..n}:
   unknowns := {x[i] $ i = 1..n}:
   linsolve(equations, unknowns)
```

```
 --         4 x[0]   x[5]              3 x[0]   2 x[5]
 |  x[1] = ------ + ---- - 2, x[2] = ------ + ------ - 3,
 --           5       5                  5        5

            2 x[0]   3 x[5]               x[0]   4 x[5]      --
    x[3] = ------ + ------ - 3, x[4] = ---- + ------ - 2  |
              5        5                  5       5         --
```

Symbolic indexed objects are of type `"_index"`:

```
>> type(x[i])

                           "_index"

>> delete n, equations, unknowns:
```

Example 2. Lists, arrays and tables are typical containers allowing indexed access to their elements:

```
>> L := [1, 2, [3, 4]]:
   A := array(1..2, 2..3, [[a12, a13], [a22, a23]]):
   T := table( 1 = T1, x = Tx, (1, 2) = T12):

>> L[1], L[3][2], A[2, 3], T[1], T[x], T[1, 2]
```

```
                    1, 4, a23, T1, Tx, T12
```

The entries can be changed via indexed assignments:

```
>> L[2] := 22: L[3][2]:= 32: A[2, 3]:= 23: T[x] := T12: L, A, T

                          +-          -+  table(
                          |  a12, a13  |    (1, 2) = T12,
        [1, 22, [3, 32]], |            |,   x = T12,
                          |  a22,  23  |    1 = T1
                          +-          -+  )

>> delete L, A, T:
```

Example 3. For finite sets, an indexed call `x[i]` returns the `i`-th element as printed on the screen. This element does not necessarily coincide with the `i`-th (internal) operand as returned by `op`:

```
>> S := {1, 2, 3, x}

                          {x, 1, 2, 3}

>> S[i] $ i = 1..4

                           x, 1, 2, 3

>> op(S, i) $ i = 1..4

                           x, 3, 2, 1

>> delete S:
```

Example 4. The index operator also operates on character strings. Note that the characters are enumerated from 0:

```
>> "ABCDEF"[0], "ABCDEF"[5]

                            "A", "F"
```

Example 5. Indexed calls evaluate the returned entry:

```
>> delete a: x := [a, b]: a := c: x[1] = eval(x[1])

                              c = c

>> delete a: x := table(1 = a, 2 = b): a := c:
   x[1] = eval(x[1]), x[1] <> indexval(x, 1)

                          c = c, c <> a

>> delete a: x := array(1..2, [a, b]): a := c:
   x[1] = eval(x[1]), x[1] <> indexval(x, 1)

                          c = c, c <> a

>> delete a: x := matrix([a, b]): a := c: x[1] = eval(x[1])

                              c = c

>> delete x, a:
```

Changes:

- Indexed access to the elements of matrices now evaluates the returned entry.

`intersect`, `minus`, `union` – operators for sets and intervals

`intersect` computes the intersection of sets and intervals.

`minus` computes the difference between sets and intervals.

`union` computes the union of sets and intervals.

Call(s):

- `set1 intersect set2`
- `_intersect(set1, set2, ...)`
- `set1 minus set2`
- `_minus(set1, set2)`
- `set1 union set2`
- `_union(set1, set2, ...)`

Parameters:

`set1, set2, ...` — finite sets of type `DOM_SET`, or intervals of type `Dom::Interval`, or arithmetical expressions

Return Value: a set, an interval, a symbolic expression of type `"_intersect"`, `"_minus"`, `"_union"`, or `universe`.

Overloadable by: `set1, set2, ...`

Related Functions: `universe`

Details:

- `set1 intersect set2` is equivalent to `_intersect(set1, set2)`.
- `set1 minus set2` is equivalent to `_minus(set1, set2)`.
- `set1 union set2` is equivalent to `_union(set1, set2)`.
- The precedences of `intersect`, `minus`, `union` are as follows: The operator `intersect` is stronger binding than `minus`, i.e,

  `set1 intersect set2 minus set3` = `(set 1 intersect set2) minus set3`.

  The operator `minus` is stronger binding than `union`, i.e.,

  `set1 minus set2 union set3` = `(set1 minus set2) union set3`.

  Further,

  `set1 minus set2 minus set3` = `(set 1 minus set2) minus set3`.

  If in doubt, use brackets to make sure that the expression is parsed as desired.
- If sets or intervals are specified by symbolic expressions involving identifiers or indexed identifiers, then symbolic calls of `_intersect`, `_minus`, `_union` are returned. On the screen, they are represented via the operator notation `set1 intersect set2` etc.
- On finite sets of type `DOM_SET`, these operators act in a purely *syntactical* way. E.g., `{1} minus {x}` simplifies to `{1}`. Mathematically, this result may not be correct in general, because `x` might represent the value 1.
- On intervals of type `Dom::Interval`, these operators act in a *semantical* way. In particular, properties of identifiers are taken into account.

- `_intersect()` returns `universe` (of type `stdlib::Universe`) which represents the set of all mathematical objects.
- `_union()` returns the empty set `{}`.
- `_intersect` is a function of the system kernel.
- `_minus` is a function of the system kernel.
- `_union` is a function of the system kernel.

Example 1. `intersect`, `minus`, and `union` operate on finite sets:

```
>> {x, 1, 5} intersect {x, 1, 3, 4},
   {x, 1, 5} union {x, 1, 3, 4},
   {x, 1, 5} minus {x, 1, 3, 4}

                  {x, 1}, {x, 1, 3, 4, 5}, {5}
```

For symbolic sets, specified as identifiers or indexed identifiers, symbolic calls are returned:

```
>> {1, 2} union A union {2, 3}

                        {1, 2, 3} union A
```

Note that the set operations act on finite sets in a purely syntactical way. In the following call, `x` does not match any of the numbers 1, 2, 3 syntactically:

```
>> {1, 2, 3} minus {1, x}

                             {2, 3}
```

Example 2. `intersect`, `minus`, and `union` are overloaded by the domain `Dom::Interval`:

```
>> Dom::Interval([0, 1]) union Dom::Interval(1, 4)

                             [0, 4[

>> Dom::Interval([0, 1]) union Dom::Interval(4, infinity)

                   [0, 1] union ]4, infinity[

>> Dom::Interval(2, infinity) intersect Dom::Interval([1, 3])

                             ]2, 3]

>> {PI/2, 2, 2.5, 3} intersect Dom::Interval(1,3)
```

```
{         PI }
{ 2.5, 2, -- }
{          2 }
```

```
>> Dom::Interval(1, PI) minus {2, 3}
```

```
]3, PI[ union ]1, 2[ union ]2, 3[
```

In contrast to finite sets of type `DOM_SET`, the interval domain works semantically. It takes `properties` into account:

```
>> Dom::Interval(-1, 1) minus {x}
```

```
]x, 1[ union ]-1, x[
```

```
>> assume(x > 2): Dom::Interval(-1, 1) minus {x}
```

```
]-1, 1[
```

```
>> unassume(x):
```

Example 3. The following list provides a collection of sets:

```
>> L := [{a, b}, {1, 2, a, c}, {3, a, b}, {a, c}]:
```

The functional equivalent `_intersect` of the `intersect` operator accepts an arbitray number of arguments. Thus, the intersection of all sets in `L` can be computed as follows:

```
>> _intersect(op(L))
```

```
{a}
```

The union of all sets in `L` is:

```
>> _union(op(L))
```

```
{a, b, c, 1, 2, 3}
```

```
>> delete L:
```

Example 4. `universe` represents the set of all mathematical objects:

```
>> _intersect()
```

```
universe
```

`_invert` – the reciprocal of an expression

`_invert(x)` computes the reciprocal `1/x` of `x`.

Call(s):

- `1/x`
- `_invert(x)`

Parameters:

`x` — an arithmetical expression or a set

Return Value: an arithmetical expression or a set.

Overloadable by: `x`

Related Functions: `_divide`, `_negate`, `^`, `/`, `*`, `+`, `-`

Details:

- `1/x` is equivalent to the function call `_invert(x)`. It represents the inverse of the element `x` with respect to multiplication, i.e., `x * (1/x) = 1`.
- The reciprocal of a number of type `Type::Numeric` is returned as a number.
- `1/x` is overloaded for matrix domains (`matrix`) and returns the inverse of the matrix `x`.
- If `x` is not an element of a library domain with an `"_invert"` method, `1/x` is internally represented as `x^(-1)` = `_power(x, -1)`.
- If `x` is an element of a domain with a slot `"_invert"`, then this method is used to compute `1/x`. Many library domains overload the `/` operator by an appropriate `"_invert"` slot. Note that `a/x` calls the overloading slot `x::dom::_invert(x)` only for `a = 1`.
- If neither `x` nor `y` overload the binary operator `/` by a `"_divide"` method, the quotient `x/y` is equivalent to `x * y^(-1)` = `_mult(x, _power(y, -1))`.
- For finite sets, `1/X` is the set $\{1/x;\ x \in X\}$.
- `_invert` is a function of the system kernel.

Example 1. The reciprocal of an expression is the inverse with respect to `*`:

```
>> _invert(x), x * (1/x) = x * _invert(x)

                              1
                              -, 1 = 1
                              x
```

```
>> 3 * y * x^2 / 27 / x

                                    x y
                                    ---
                                     9
```

Internally, a symbolic expression `1/x` is represented as `x^(-1)` = `_power(x, -1)`:

```
>> type(1/x), op(1/x, 0), op(1/x, 1), op(1/x, 2)

                      "_power", _power, x, -1
```

Example 2. For finite sets, `1/X` is the set $\{1/x;\ x \in X\}$:

```
>> 1/{a, b, c}

                                { 1  1  1 }
                                { -, -, - }
                                { a  b  c }
```

Example 3. Various library domains such as matrix domains or residue class domains overload `_invert`:

```
>> x := Dom::Matrix(Dom::IntegerMod(7))([[2, 3], [3, 4]]):
   x, 1/x, x * (1/x)

 +-                 -+  +-                 -+
 |  2 mod 7, 3 mod 7  |  |  3 mod 7, 3 mod 7  |
 |                    |, |                    |,
 |  3 mod 7, 4 mod 7  |  |  3 mod 7, 5 mod 7  |
 +-                 -+  +-                 -+

    +-                 -+
    |  1 mod 7, 0 mod 7  |
    |                    |
    |  0 mod 7, 1 mod 7  |
    +-                 -+

>> delete x:
```

`_lazy_and, _lazy_or` – "lazy evaluation" of Boolean expressions

`_lazy_and(b1, b2, ...)` evaluates the Boolean expression `b1 and b2 and ...` by "lazy evaluation".

`_lazy_or(b1, b2, ...)` evaluates the Boolean expression `b1 or b2 or ...` by "lazy evaluation".

Call(s):

- `_lazy_and(b1, b2, ...)`
- `_lazy_or(b1, b2, ...)`

Parameters:

`b1, b2, ...` — Boolean expressions

Return Value: `TRUE`, `FALSE`, or `UNKNOWN`.

Overloadable by: `b1, b2, ...`

Related Functions: `and`, `bool`, `if`, `is`, `or`, `repeat`, `while`, `FALSE`, `TRUE`, `UNKNOWN`

Details:

- `_lazy_and(b1, b2, ...)` produces the same result as `bool(b1 and b2 and ...)`, provided the latter call does not produce an error. The difference between these calls is as follows:

  `bool(b1 and b2 and ...)` evaluates *all* Boolean expressions before combining them logically via 'and'.

  Note that the result is `FALSE` if one of `b1`, `b2` etc. evaluates to `FALSE`. "Lazy evaluation" is based on this fact: `_lazy_and(b1, b2, ...)` evaluates the arguments from left to right. The evaluation is stopped immediately if one argument evaluates to `FALSE`. In this case, `_lazy_and` returns `FALSE` *without* evaluating the remaining Boolean expressions. If none of the expressions `b1`, `b2` etc. evaluates to `FALSE`, then all arguments are evaluated and the corresponding result `TRUE` or `UNKNOWN` is returned.

  `_lazy_and` is also called "conditional `and`".

- `_lazy_or(b1, b2, ...)` produces the same result as `bool(b1 or b2 or ...)`, provided the latter call does not produce an error. The difference between these calls is as follows:

  `bool(b1 or b2 or ...)` evaluates *all* Boolean expressions before combining them logically via 'or'.

  Note that the result is `TRUE` if one of `b1`, `b2` etc. evaluates to `TRUE`. "Lazy evaluation" is based on this fact: `_lazy_or(b1, b2, ...)` evaluates the arguments from left to right. The evaluation is stopped immediately if one argument evaluates to `TRUE`. In this case, `_lazy_or` returns `TRUE` *without* evaluating the remaining Boolean expressions. If none of the expressions `b1`, `b2` etc. evaluates to `TRUE`, then all arguments are evaluated and the corresponding result `FALSE` or `UNKNOWN` is returned.

`_lazy_or` is also called "conditional `or`".

- If any of the considered Boolean expressions `b1`, `b2` etc. cannot be evaluated to `TRUE`, `FALSE`, or `UNKNOWN`, then `_lazy_and`, `_lazy_or` produce errors.
- `_lazy_and` and `_lazy_or` are internally used by the `if`, `repeat`, and `while` statements. For example, the statement '`if b1 and b2 then ...`' is equivalent to '`if _lazy_and(b1, b2) then ...`'.
- `_lazy_and()` returns `TRUE`.
- `_lazy_or()` returns `FALSE`.
- `_lazy_and` is a function of the system kernel.
- `_lazy_or` is a function of the system kernel.

Example 1. This example demonstrates the difference between lazy evaluation and complete evaluation of Boolean conditions. For $x = 0$, the evaluation of $\sin(1/x)$ leads to an error:

```
>> x := 0: bool(x <> 0 and sin(1/x) = 0)

 Error: Division by zero
```

With "lazy evaluation", the expression $\sin(1/x) = 0$ is not evaluated. This avoids the previous error:

```
>> _lazy_and(x <> 0, sin(1/x) = 0)

                              FALSE

>> bool(x = 0 or sin(1/x) = 0)

 Error: Division by zero

>> _lazy_or(x = 0, sin(1/x) = 0)

                              TRUE

>> delete x:
```

Example 2. The following statements do no produce an error, because `if` uses lazy evaluation internally:

```
>> for x in [0, PI, 1/PI] do
     if x = 0 or sin(1/x) = 0 then
        print(x)
     end_if;
   end_for:
```

```
                                0

                                1
                               --
                               PI
```

```
>> delete x:
```

Example 3. Both functions can be called without parameters:

```
>> _lazy_and(), _lazy_or()
```

```
                          TRUE, FALSE
```

`_negate` – the negative of an expression

`_negate(x)` computes the negative of `x`.

Call(s):

- `-x`
- `_negate(x)`

Parameters:

`x` — an arithmetical expression, a polynomial of type `DOM_POLY`, or a set

Return Value: an arithmetical expression, a polynomial, or a set.

Overloadable by: `x`

Related Functions: `_invert`, `_subtract`, `^`, `/`, `*`, `+`, `-`, `poly`

Details:

- `-x` is equivalent to the function call `_negate(x)`. It represents the inverse of the element `x` of an additive group. For standard expressions, `-x` is the inverse with respect to the `+` operation.
- The negative of a number of type `Type::Numeric` is returned as a number.
- If `x` is not an element of a libary domain with a `"_negate"` method, `-x` is internally represented as `x*(-1)` = `_mult(x, -1)`.
- If `x` is an element of a domain with a slot `"_negate"`, then this method is used to compute `-x`. Many library domains overload the unary `-` operator by an appropriate `"_negate"` slot.
- If neither `x` nor `y` overload the *binary* operator `-` by a `"_subtract"` method, the difference `x - y` is equivalent to `x + y*(-1)` = `_plus(x, _mult(y, -1))`.
- The negative of a polynomial of type `DOM_POLY` yields a polynomial with the negative of the original coefficients.
- For finite sets, `-X` is the set $\{-x;\ x \in X\}$.
- `_negate` is a function of the system kernel.

Example 1. The negative of an expression is the inverse with respect to `+`:

```
>> x - x = x + _negate(x)

                          0 = 0

>> -1 + x - 2*x + 23

                         22 - x
```

Internally, a symbolic `-x` is represented as `x*(-1)` = `_mult(x, -1)`:

```
>> type(-x), op(-x, 0), op(-x, 1), op(-x, 2)

                  "_mult", _mult, x, -1
```

Example 2. The negative of a polynomial yields a polynomial:

```
>> -poly(x^2 + x - 1, [x])

                          2
                 poly(- x  - x + 1, [x])

>> -poly(x, [x], Dom::Integer)

             poly((-1) x, [x], Dom::Integer)
```

Example 3. For finite sets, `-X` is the set $\{-x;\ x \in X\}$:

```
>> -{a, b, c}

                           {-a, -b, -c}
```

Example 4. Various library domains such as matrix domains or residue class domains overload `_negate`:

```
>> x := Dom::Matrix(Dom::IntegerMod(7))([2, 10]): x, -x, x + (-x)

           +-          -+  +-          -+  +-          -+
           |  2 mod 7   |  |  5 mod 7   |  |  0 mod 7   |
           |            |, |            |, |            |
           |  3 mod 7   |  |  4 mod 7   |  |  0 mod 7   |
           +-          -+  +-          -+  +-          -+

>> delete x:
```

Example 5. This example demonstrates how to implement a slot `"_negate"` for a domain. The following domain `myString` is to represent character strings. The negative `-x` of such a string `x` is to consist of the characters in reverse order.

The `"new"` method uses `expr2text` to convert any MuPAD object to a string. This string is the internal representation of elements of `myString`. The `"print"` method turns this string into the screen output:

```
>> myString := newDomain("myString"):
   myString::new := proc(x)
   begin
     if args(0) = 0 then x := "" end_if;
     case domtype(x)
       of myString do return(x);
       of DOM_STRING do return(new(dom, x));
       otherwise return(new(dom, expr2text(x)));
     end_case
   end_proc:
   myString::print := x -> extop(x, 1):
```

Without a `"_negate"` method, the system handles elements of this domain like any symbolic object:

```
>> x := myString(x): -x, type(-x), op(-x, 0), op(-x, 1), op(-x, 2)

                    -x, "_mult", _mult, x, -1
```

Now, we implement the "_negate" method. There is no need to check the argument, because _negate(x) calls this slot if and only if x is of type myString. The slot uses revert to generate the reverted string:

```
>> myString::_negate := x -> myString::new(revert(extop(x, 1))):
```

Now, myString objects can be reverted by the - operator:

```
>> -myString("This is a string")

                        gnirts a si sihT
```

In the following call, myString::_negate is not called because there is no "_subtract" method for myString objects:

```
>> myString("This is a string") - myString("a string")

                   This is a string - a string
```

We provide the slots "_plus" and "_subtract":

```
>> myString::_plus := proc()
   begin
     myString::new(_concat(map(args(), extop, 1))):
   end_proc:
   myString::_subtract := (x, y) -> x + myString::_negate(y):
```

Now, the "_negate" slot is called:

```
>> myString("This is a string") - myString("This is a string")

                This is a stringgnirts a si sihT
```

```
>> delete myString, x:
```

_stmtseq – statement sequences

The function call _stmtseq(object1, object2, ...) is equivalent to the statement sequence (object1; object2; ...).

Call(s):

- (object1; object2; ...)
- (object1: object2: ...)
- _stmtseq(object1, object2, ...)

Parameters:

`object1, object2, ...` — arbitrary MuPAD objects and statements

Return Value: the return value of the last statement in the sequence.

Related Functions: `_exprseq`

Details:

- The function call `_stmtseq(object1, object2, ...)` evaluates the statements `(object1; object2; ...)` from left to right.
- `_stmtseq()` returns the void object of type `DOM_NULL`.
- `_stmtseq` is a function of the system kernel.

Example 1. Usually, statements are entered imperatively:

```
>> x := 2; x := x^2 + 17; sin(x + 1)

                                 2

                                 21

                               sin(22)
```

This sequence of statements is turned into a single command (a "statement sequence") by enclosing it in brackets. Now, only the result of the "statement sequence" is printed. It is the result of the last statement inside the sequence:

```
>> (x := 2; x := x^2 + 17; sin(x + 1))

                               sin(22)
```

Alternatively, the statement sequence can be entered via `_stmtseq`. For syntactical reasons, the assignments have to be enclosed in brackets when using them as arguments for `_stmtseq`. Only the return value of the statement sequence (the return value of the last statement) is printed:

```
>> _stmtseq((x := 2), (x := x^2 + 17),  sin(x + 1))

                               sin(22)
```

Statement sequences can be iterated:

```
>> x := 1: (x := x + 1; x := x^2; print(i, x)) $ i = 1..4
```

```
1, 4

2, 25

3, 676

4, 458329
```

```
>> delete x:
```

`%if` – conditional creation of code by the parser

`%if` controls the creation of code by the parser depending on a condition.

Call(s):

- `%if condition`
  `then casetrue`
  `<elif condition then casetrue, ...>`
  `<else casefalse>`
  `end_if`

Parameters:

`condition` — a Boolean expression
`casetrue` — a statement sequence
`casefalse` — a statement sequence

Related Functions: `if`

Details:

- This statement is one of the more esoteric features of MuPAD. It is *not* executed at run time by the MuPAD interpreter. It controls the creation of code for the interpreter by the parser.
- `%if` may be used to create different versions of a library which share a common code basis, or to insert debugging code which should not appear in the release version.
- The first condition is executed by the parser in a Boolean context and must yield one of the Boolean values `TRUE` or `FALSE`:
  - If the condition yields `TRUE`, the statement sequence `casetrue` is the code that is created by the parser for the `%if`-statement. The rest of the statement is ignored by the parser, no code is created for it.

- If the condition yields `FALSE`, then the condition of the next `elif`-part if evaluated and the parser continues as before.
- If all conditions evaluate to `FALSE` and no more `elif`-parts exist, the parser inserts the code of the statement sequence `casefalse` as the code for the `%if`-statement. If no `casefalse` exists, `NIL` is produced.

The whole statement sequence is read by the parser and must be syntactically correct. Also the parts that do not result in code must be syntactically correct.

Note that instead of `end_if`, one may also simply use the keyword `end`.

In case of an empty statement sequence, the parser creates `NIL` as code.

NOTE

The conditions are parsed in the lexical context where they occur, but are *evaluated by the parser in the context where the parser is executed*. This is the case because the environment where the conditions are lexically bound simply does not exist during parsing. Thus, one must ensure that names in the conditions do not conflict with names of local variables or arguments in the surrounding lexical context. The parser does not check this!

No function exists in the interpreter which can execute the `%if`-statement. The reason is that the statement is implemented by the parser, not by the interpreter.

Example 1. In the following example, we create debugging code in a procedure depending on the value of the global identifier `DEBUG`.

Note that this example is somewhat academic, as the function `prog::trace` is a much more elegant way to trace a procedure during debugging.

```
>> DEBUG := TRUE:
   p := proc(x) begin
        %if DEBUG = TRUE then
            print("entering p")
        end;
        x^2
   end_proc:
   p(2)

                         "entering p"

                              4
```

When we look at `p`, we see that only the `print` command was inserted by the parser:

```
>> expose(p)
```

```
proc(x)
  name p;
begin
  print("entering p");
  x^2
end_proc
```

Now we set `DEBUG` to `FALSE` and parse the procedure again to create the release version. No debug output is printed:

```
>> DEBUG := FALSE:
   p := proc(x) begin
        %if DEBUG = TRUE then
            print("entering p")
        end;
        x^2
   end_proc:
   p(2)
```

```
4
```

If we look at the procedure we see that `NIL` was inserted for the `%if`-statement:

```
>> expose(p)
```

```
proc(x)
  name p;
begin
  NIL;
  x^2
end_proc
```

Background:

- This statement may remind C programmers of conditional compiliaton. In C, this is implemented by a pre-processor which is run before the parser. In MuPAD, such a pre-processor does not exist. The `%if`-statement is part of the parsing process.

`Ci` – the cosine integral function

`Ci(x)` represents the cosine integral `EULER` $+ \ln(x) + \int_0^x (\cos(t) - 1)/t \; dt$.

Call(s):

- `Ci(x)`

Parameters:

`x` — an arithmetical expression

Return Value: an arithmetical expression.

Overloadable by: `x`

Side Effects: When called with a floating point argument, the function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Related Functions: `Ei`, `int`, `Si`, `cos`

Details:

- If `x` is a floating point number, then `Ci(x)` returns the numerical value of the cosine integral. The special values $Ci(\infty) = 0$ and $Ci(-\infty) = i\,\pi$ are implemented. For all other arguments, `Ci` returns a symbolic function call.
- The float attribute of `Ci` is a kernel function, i.e., floating point evaluation is fast.

Example 1. We demonstrate some calls with exact and symbolic input data:

```
>> Ci(1), Ci(sqrt(2)), Ci(x + 1), Ci(infinity), Ci(-infinity)

                             1/2
                Ci(1), Ci(2     ), Ci(x + 1), 0, I PI
```

Floating point values are computed for floating point arguments:

```
>> Ci(1.0), Ci(2.0 + 10.0*I)

            0.3374039229, - 242.5252694 - 1185.8387 I
```

Example 2. `Ci` is singular at the origin:

```
>> Ci(0)

 Error: singularity [Ci]
```

The negative real axis is a branch cut of `Ci`. A jump of height $2\pi i$ occurs when crossing this cut:

```
>> Ci(-1.0), Ci(-1.0 + 10^(-10)*I), Ci(-1.0 - 10^(-10)*I)

 0.3374039229 + 3.141592654 I, 0.3374039229 + 3.141592654 I,

    0.3374039229 - 3.141592654 I
```

Example 3. The functions `diff`, `float`, and `series` handle expressions involving `Ci`:

```
>> diff(Ci(x), x, x, x), float(ln(3 + Ci(sqrt(PI))))

            2 cos(x)   cos(x)   2 sin(x)
            -------- - ------ + --------, 1.241299561
                3        x          2
               x                   x

>> series(Ci(x), x = 0);
   series(Ci(x), x = infinity, 5);

                                   2    4
                                  x    x       6
                (ln(x) + EULER) - -- + -- + O(x )
                                   4   96

   sin(x)   cos(x)   2 sin(x)   6 cos(x)   24 sin(x)    /  1 \
   ------ - ------ - -------- + -------- + --------- + O| -- |
     x        2         3          4           5        |  6 |
             x         x          x           x         \ x  /
```

Background:

- The function `Ci(x)-ln(x)` is an entire function. `Ci` has a logarithmic singularity at the origin and a branch cut along the negative real axis. The values on the negative real axis coincide with the limit "from above":

$$Ci(x) = \lim_{\epsilon \to 0_+} Ci(x + \epsilon\, i)\ , \quad x \text{ real, } x < 0.$$

- Reference: M. Abramowitz and I. Stegun, "Handbook of Mathematical Functions", Dover Publications Inc., New York (1965).

Changes:

- Series expansions can now be computed via `series`.

`D` – differential operator for functions

`D(f)` or, alternatively, `f'` computes the derivative of the univariate function `f`.

`D([n1, n2, ...], f)` computes the partial derivative $\frac{\partial}{\partial x_{n_1}} \frac{\partial}{\partial x_{n_2}} \cdots f$ of the multivariate function $f(x_1, x_2, \dots)$.

Call(s):

- `f'`
- `D(f)`
- `D([n1, n2, ...], f)`

Parameters:

`f`	—	a function or a functional expression, an array, a list, a polynomial, a set, or a table
`n1, n2, ...`	—	indices: positive integers

Return Value: a function or a functional expression. If `f` is an array or a list etc., a corresponding object containing the derivatives of the entries is returned.

Side Effects: `D` uses `option remember`.

Overloadable by: `f`

Further Documentation: Section 7.1 of the MuPAD Tutorial.

Related Functions: `diff`, `int`, `poly`

Details:

- MuPAD has two functions for differentiation: `diff` and `D`. `D` represents the differential operator that may be applied to *functions*; `diff` is used to differentiate *arithmetical expressions*. Mathematically, `D(f)(x)` coincides with `diff(f(x), x)`; `D([1, 2], f)(x, y)` coincides with `diff(f(x, y), x, y)`. Symbolic calls of `D` and `diff` can be converted to one another via `rewrite`. Cf. example 8.

- `D(f)` returns the derivative f' of the univariate function `f`. `f'` is shorthand for `D(f)`.

- If f is a multivariate function and $D_n f$ denotes the partial derivative of f with respect to its n-th argument, then `D([n1, n2, ...], f)` computes the partial derivative $D_{n_1} D_{n_2} \cdots f$. Cf. example 5. In particular, `D([ ], f)` returns f itself.

  It is assumed that partial derivatives commute. Internally, `D([n1, n2, ...], f)` is converted to `D([m1, m2, ...], f)`, where `[m1, m2, ...]` = `sort([n1, n2, ...])`.

- f may be any object which can represent a function. In particular, f may be a functional expression built from simple functions by means of arithmetic operators (+, -, *, /, ^, @, @@). Any identifier different from CATALAN, EULER, and PI is regarded as an "unknown" function; the same holds for elements of kernel domains not explicitly mentioned on this page. Cf. example 1. Any number and each of the three constant identifiers above is regarded as a constant function. Cf. example 2.

- If f is a list, a set, a table, or an array, then D is applied to each entry of f. Cf. example 3.

- A polynomial f of type DOM_POLY is regarded as polynomial function, the indeterminates being the arguments of the function. Cf. example 6.

- If f is a function environment, a procedure, or a builtin kernel function, then D can compute the derivative in some cases; see the "Background" section below. If this is not possible, a symbolic D call is returned.

- Higher partial derivatives `D([n1], D([n2], f))` are simplified to `D([n1, n2], f)`. Cf. example 7.

- The derivative of a univariate function f —denoted by `D(f)`— is syntactically distinguished from the partial derivative `D([1], f)` with respect to the first variable, even if f represents a univariate function.

- The usual rules of differentiation are implemented:

  - `D(f + g) = D(f) + D(g)`,
  - `D(f * g) = f * D(g) + g * D(f)`,
  - `D(1/f) = -D(f) / f^2`,
  - `D(f @ g) = D(f) @ g` * D(g).

  Note that the composition of functions is written as `f@g` and *not* as `f(g)`.

- In order to express the n-th derivative of a univariate function for symbolic n, you can use the "repeated composition operator" @@. Cf. example 9.

Example 1. `D(f)` computes the derivative of the function `f`:

```
>> D(sin), D(x -> x^2), D(id)

                        cos, 2 id, 1
```

Note that `id` denotes the identity function. `D` also works for more complex functional expressions:

```
>> D(sin @ exp + 2*(x -> x*ln(x)) + id^2)

                 2 id + 2 ln + exp cos@exp + 2
```

If `f` is an identifier without a value, a symbolic `D` call is returned:

```
>> delete f: D(f + sin)

                         D(f) + cos
```

The same holds for objects of kernel type that cannot be regarded as functions:

```
>> D(NIL), D(point(3,2))

                   D(NIL), D(point(3, 2))
```

`f'` is shorthand for `D(f)`:

```
>> (f + sin)', (x -> x^2)', id'

                     D(f) + cos, 2 id, 1
```

Example 2. Constants are regarded as constant functions:

```
>> PI', 3', (1/2)'

                           0, 0, 0
```

Example 3. The usual rules of differentiation are implemented. Note that lists and sets may also be taken as input; in this case, `D` is applied to each element of the list or set:

```
>> delete f, g: D([f+g, f*g]); D({f@g, 1/f})

                [D(f) + D(g), f D(g) + g D(f)]

                   {                D(f) }
                   { D(g) D(f)@g, - ---- }
                   {                  2  }
                   {                 f   }
```

Example 4. The derivatives of most special functions of the library can be computed. Again, `id` denotes the identity function:

```
>> D(tan); D(sin*cos); D(1/sin); D(sin@cos); D(2*sin + ln)

                                 2
                              tan  + 1

                                2      2
                             cos  - sin

                                  cos
                                - ----
                                     2
                                  sin

                            -sin cos@cos

                              1
                             -- + 2 cos
                             id
```

Example 5. `D` can also compute derivatives of procedures:

```
>> f := x -> x^2:
   g := proc(x) begin tan(ln(x)) end:
   D(f), D(g)

                                    2
                              tan@ln  + 1
                        2 id, -----------
                                   id
```

We differentiate a function of two arguments by passing a list of indices as first argument to `D`. In the example below, we first differentiate with respect to the second argument and then differentiate the result with respect to the first argument:

```
>> D([1, 2], (x, y) -> sin(x*y))

                (x, y) -> cos(x*y) - x*y*sin(x*y)
```

The order of the partial derivatives is not relevant:

```
>> D([2, 1], (x, y) -> sin(x*y))

                (x, y) -> cos(x*y) - x*y*sin(x*y)

>> delete f, g:
```

Example 6. A polynomial is regarded as a polynomial function:

```
>> D(poly(x^2 + 3*x + 2, [x]))
```

```
                    poly(2 x + 3, [x])
```

We differentiate the following bivariate polynomial `f` twice with respect to its second variable `y` and once with respect to its first variable `x`:

```
>> f := poly(x^3*y^3, [x, y]):
   D([1, 2, 2], f) = diff(f, y, y, x)
```

```
                 2                        2
        poly(18 x  y, [x, y]) = poly(18 x  y, [x, y])
```

```
>> delete f:
```

Example 7. Nested calls to `D` are flattened:

```
>> D([1], D([2], f))
```

```
                       D([1, 2], f)
```

However, this does not hold for calls with only one argument, since `D(f)` and `D([1], f)` are not considered to be the same:

```
>> D(D(f))
```

```
                          D(D(f))
```

Example 8. `D` may only be applied to functions whereas `diff` makes only sense for expressions:

```
>> D(sin), diff(sin(x), x)
```

```
                        cos, cos(x)
```

Applying `D` to expressions and `diff` to functions makes no sense:

```
>> D(sin(x)), diff(sin, x)
```

```
                       D(sin(x)), 0
```

`rewrite` allows to rewrite expressions with `D` into `diff`-expression:

```
>> rewrite(D(f)(y), diff), rewrite(D(D(f))(y), diff)
```

```
               diff(f(y), y), diff(f(y), y, y)
```

The reverse conversion is possible as well:

```
>> map(%, rewrite, D)
```

```
                    D(f)(y), D(D(f))(y)
```

Example 9. Sometimes you may need the n-th derivative of a function, where n is unknown. This can be achieved using the repeated composition operator. For example, let us write a function that computes the k-th Taylor polynomial of a function f at a point x_0 and uses x as variable for that polynomial:

```
>> kthtaylorpoly:=
   (f, k, x, x0) -> _plus(((D@@n)(f)(x0) * (x - x0)^n / n!) $ n = 0..k):
   kthtaylorpoly(sin, 7, x, 0)

                                      3     5      7
                                     x     x      x
                                 x - -- + --- - ----
                                     6    120   5040

>> delete kthtaylorpoly:
```

Example 10. Advanced users can extend `D` to their own special mathematical functions (see "Background" section below). To this end, embed your mathematical function f, say, into a function environment `f` and implement the behavior of `D` for this function as the `"D"` slot of the function environment. The slot must handle two cases: it may be either called with only one argument which equals `f`, or with two arguments where the second one equals `f`. In the latter case, the first argument is a list of arbitrary many indices; that is, the slot must be able to handle higher partial derivatives also.

Suppose, for example, that we are given a function $f(t, x, y)$, and that we do not know anything about f except that it is differentiable infinitely often and satisfies the partial differential equation $\frac{\partial f}{\partial t} = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$. To make MuPAD eliminate derivatives with respect to `t`, we can do the following:

```
>> f := funcenv(f):
   f::D :=
   proc(indexlist, ff)
     local
       n         : DOM_INT,   // Number of t-derivatives.
       list_2_3  : DOM_LIST;  // List of indices of 2's and 3's.
                              // These remain unchanged.
   begin
     if args(0) <> 2 then
       error("Wrong number of arguments")
     end_if;
     n        :=  nops(select(indexlist, _equal, 1));
     list_2_3 :=  select(indexlist, _unequal, 1);
     // rewrite (d/dt)^n = (d^2/dx^2 + d^2/dy^2)^n
     _plus(binomial(n, k) *
           hold(D)(sort([2 $ 2*(n-k), 3 $ 2*k].list_2_3), ff)
           $ k = 0..n)
   end_proc:
```

Now, partial derivatives with respect to the first argument t are rewritten by derivatives with respect to the second and third argument:

```
>> D([1], f^2)(t, x, y)

  2 f(t, x, y) (D([2, 2], f)(t, x, y) + D([3, 3], f)(t, x, y))

>> D([1, 2, 1], f)

 D([2, 2, 2, 2, 2], f) + 2 D([2, 2, 2, 3, 3], f) +

    D([2, 3, 3, 3, 3], f)

>> delete f:
```

Background:

- If `f` is a domain or a function environment with a slot `"D"`, this slot is called to compute the derivative. The slot procedure has the same calling syntax as `D`. In particular —and in contrast to the slot `"diff"`— the slot must be able to compute higher partial derivatives because the list of indices may have length greater than one. Cf. example 10.
- If `f` is a procedure, a function environment without a `"D"` slot, or a builtin kernel function (an "executable object"), then `f` is called with auxiliary identifiers as arguments. The result of the call is then differentiated using the function `diff`. If the result of `diff` yields an expression which can be regarded as function in the auxiliary identifers, then this function is returned, otherwise an unevaluated call of `D` is returned.
- Let us take the function environment `sin` as an example. It has no `"D"` slot, thus the procedure `op(sin, 1)`, which is responsible for evaluating the sine function, is used to compute `D(sin)`, as follows. This procedure is applied to an auxiliary identifier, say `x`, and differentiated with respect to this identifier via `diff`. The result is `diff(sin(x), x) = cos(x)`. Via `fp::expr_unapply` and `fp::unapply`, the function `cos` is computed as the derivative of `sin`.

Changes:

- Assuming that partial derivatives commute, the ordering of higher partial derivatives is now "normalized". E.g., `D([2, 1], f)` is now automatically rewritten as `D([1, 2], f)`.
- `D` now uses `option remember`.

`DIGITS` – the significant digits of floating point numbers

The environment variable `DIGITS` determines the number of significant decimal digits in floating point numbers. The default value is `DIGITS` = 10.

Call(s):

- `DIGITS`
- `DIGITS := n`

Parameters:

`n` — a positive integer smaller than 2^{31}.

Related Functions: `float`, `Pref::floatFormat`, `Pref::trailingZeroes`

Details:

- Floating point numbers are created by applying the function `float` to exact numbers or numerical expressions. Elementary objects are approximated by the resulting floats with a relative precision of `10^(-DIGITS)`, i.e., the first `DIGITS` decimal digits are correct. Cf. example 1.
- In arithmetical operations with floating point numbers, only the first `DIGITS` decimal digits are taken into account. The numerical error propagates and may grow in the course of computations. Cf. example 2.
- If a real floating point number is entered directly (e.g., by `x := 1.234`), a number with at least `DIGITS` internal decimal digits is created. Note, however, that a conversion error may occur, because the internal representation is binary.

  If a real float is entered with more than `DIGITS` digits, the internal representation stores the extra digits. However, they are not taken into account in arithmetical operations, unless `DIGITS` is increased accordingly. Cf. example 3.

  In particular, complex floating point numbers are created by adding the real and imaginary part. This addition truncates extra decimal places in the real and imaginary part.
- The value of `DIGITS` may be changed at any time during a computation. If `DIGITS` is decreased, only the leading digits of existing floating numbers are taken into account in the following arithmetical operations. If `DIGITS` is increased, existing floating point numbers are internally padded with trailing binary zeroes. Cf. example 4.

- Depending on `DIGITS`, certain functions such as the trigonometric functions may reject floats as too inaccurate and stop with an error. Cf. example 5.
- Depending on `DIGITS`, only significant digits of floating point numbers are displayed on the screen. The preferences `Pref::floatFormat` and `Pref::trailingZeroes` can be used to modify the screen output. Cf. example 4.

  At least one digit after the decimal point is displayed; if it is insignificant, it is replaced by zero. Cf. example 6.
- Internally, floating point numbers are created and stored with some extra "guard digits". These are also taken into account by the basic arithmetical operations.

  For example, for `DIGITS` = 10, the function `float` converts exact numbers to floats with about 19 decimal digits. The number of guard digits depends on `DIGITS`. For example, for all `DIGITS` from 8 through 17, the same internal representation of about 19 decimal digits is used.

  At least 2 internal guard digits are available for any value of `DIGITS`.

  Cf. examples 4 and 7.
- Environment variables such as `DIGITS` are global variables. Upon return from a procedure that changes `DIGITS`, the new value is valid outside the context of the procedure as well! Use `save DIGITS` to restrict the modified value of `DIGITS` to the procedure. Cf. example 8.
- The default value of `DIGITS` is `10`; `DIGITS` has this value after starting or resetting the system via `reset`. Also the command `delete DIGITS;` restores the default value.
- See the helppage of `float` for further information.

Example 1. We convert some exact numbers and numerical expressions to floating point approximations:

```
>> DIGITS := 10:
   float(PI), float(1/7), float(sqrt(2) + exp(3)), float(exp(-20))

  3.141592654, 0.1428571429, 21.49975049, 0.000000002061153622

>> DIGITS := 20:
   float(PI), float(1/7), float(sqrt(2) + exp(3)), float(exp(-20))

 3.1415926535897932385, 0.14285714285714285714,

    21.49975048556076279, 0.000000002061153622438557828

>> delete DIGITS:
```

Example 2. We illustrate error propagation in numerical computations. The following rational number approximates exp(2) to 17 decimal digits:

```
>> r := 738905609893065023/100000000000000000:
```

The following `float` call converts `exp(2)` and `r` to floating point approximations. The approximation errors propagate and are amplified in the following numerical expression:

```
>> DIGITS := 10: float(10^20*(r - exp(2)))

                        320.0
```

None of the digits in this result is correct. A better result is obtained by increasing `DIGITS`:

```
>> DIGITS := 20: float(10^20*(r - exp(2)))

                276.95725394785404205

>> delete r, DIGITS:
```

Example 3. In the following, only 10 of the entered 30 digits are regarded as significant. The extra digits are stored internally, anyway:

```
>> DIGITS := 10:
   a := 1.234567896666666666666666666666;
   b := 1.234567894444444444444444444444

                     1.234567897

                     1.234567894
```

We increase `DIGITS`. Because the internal representation of `a` and `b` is correct to 30 decimal place, the difference can be computed correctly to 20 decimal places:

```
>> DIGITS := 30: a - b

          0.00000000222222222222222222222

>> delete a, b, DIGITS:
```

Example 4. We compute a floating point number with a precision of 10 digits. Internally, this number is stored with about 9 guard digits to 19 correct digits. Increasing `DIGITS` to 30, the correct guard digits become visible. The remaining 11 decimal digits are created by padding the internal representation with binary zeroes. In the output, the internal representation is converted into a decimal representation. This converts the trailing binary zeroes to 11 nontrivial decimal digits. With the the call `Pref::trailingZeroes(TRUE)`, trailing zeroes of the decimal representation become visible:

```
>> DIGITS := 10: a := float(1/9)

                              0.1111111111

>> Pref::trailingZeroes(TRUE): DIGITS := 30: a

                  0.111111111111111111109605274760

>> Pref::trailingZeroes(FALSE): delete a, DIGITS:
```

Example 5. For the float evaluation of the sine function, the argument is reduced to the standard interval $[0, 2\,\pi]$. For this reduction, the argument must be known to some digits after the decimal point. For small `DIGITS`, the digits after the decimal point are pure round-off if the argument is a large floating point number:

```
>> DIGITS := 10: sin(float(2*10^20))

                              0.9576594803
```

Increasing `DIGITS` to 50, the argument of the the sine function has about 30 correct digits after the decimal point. The first 30 digits of the following result are reliable:

```
>> DIGITS := 50: sin(float(2*10^20))

      -0.98590577074208718498967738296913659461347133911129
```

For very large floating point arguments, MuPAD's trigonometric functions produce errors if `DIGITS` is not large enough:

```
>> DIGITS := 10: sin(float(2*10^30))

 Error: Loss of precision;
 during evaluation of 'sin'

>> DIGITS := 50: sin(float(2*10^30))

      0.17950046751493908795061771243112520647287791588203

>> delete DIGITS:
```

Example 6. At least one digit after the decimal point is always displayed. In the following example, the number `3.9` is displayed as `3.0` to indicate that the digit 9 after the decimal point is not significant:

```
>> DIGITS := 1: float(PI), 3.9, -3.2

                              3.0, 3.0, -3.0

>> delete DIGITS:
```

Example 7. We compute `float(10^40*8/9)` with various values of `DIGITS`. Rounding takes into account all guard digits, i.e., the resulting integer makes all guard digits visible:

```
>> for DIGITS in [7, 8, 9, 17, 18, 19, 26, 27, 28] do
     print("DIGITS" = DIGITS, round(float(10^40*8/9)))
   end_for:

    "DIGITS" = 7, 8888888887243627086483687557525021917184

    "DIGITS" = 8, 8888888888888888888303079319556646240256

    "DIGITS" = 9, 8888888888888888888303079319556646240256

    "DIGITS" = 17, 8888888888888888888303079319556646240256

    "DIGITS" = 18, 8888888888888888888888888888827804909568

    "DIGITS" = 19, 8888888888888888888888888888827804909568

    "DIGITS" = 26, 8888888888888888888888888888827804909568

    "DIGITS" = 27, 8888888888888888888888888888888888888864

    "DIGITS" = 28, 8888888888888888888888888888888888888864
```

The results show that the internal representation coincides for values of `DIGITS` between 8 and 17. Increasing `DIGITS` to 18 leads to an extended internal representation which is constant through `DIGITS` = 26. From `DIGITS` = 27 on, a yet more extended internal representation is used etc.

Example 8. The following procedure allows to compute numerical approximations with a specified precision without changing `DIGITS` as a global variable. Internally, `DIGITS` is set to the desired precision and the float approximation is computed. Because of `save DIGITS`, the value of `DIGITS` is not changed outside the procedure:

```
>> Float := proc(x, digits)
            save DIGITS;
            begin
              DIGITS := digits:
              float(x);
            end_proc:
```

The float approximation of the following value `x` suffers from numerical cancellation. In particular, for `DIGITS` = 7 only a few internal guard digits are available. The value computed by `float` has only 3 correct leading digits. `Float` is used to approximate `x` with 30 digits. The result is displayed with only 7 digits because of the value `DIGITS` = 7 valid outside the procedure. However, all displayed digits are correct:

```
>> x := PI^7 - exp(80131/10000): DIGITS := 7:
   float(x), Float(x, 30)

                    0.02779102, 0.02778943

>> delete Float, x, DIGITS:
```

Background:

- If a floating point number `x` has been created with high precision, and the computation is to continue at a lower precision, the easiest method to get rid of memory-consuming insignificant digits is `x := x + 0.0` .

`Ei` – the exponential integral function

`Ei(x)` represents the exponential integral $\int_1^\infty e^{-x\,t}/t \; dt$.

Call(s):

- `Ei(x)`

Parameters:

`x` — an arithmetical expression

Return Value: an arithmetical expression.

Overloadable by: `x`

Side Effects: When called with a floating point argument, the function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Related Functions: `Ci`, `exp`, `igamma`, `int`, `Si`

Details:

- If `x` is a floating point number, then `Ei(x)` returns the numerical value of the exponential integral. The special values $Ei(\infty) = 0$ and $Ei(-\infty) = -\infty$ are implemented. For all other arguments, `Ei` returns a symbolic function call.
- `Ei(x)` is equivalent to `igamma(0,x)` for real arguments `x>0`.
- The float attribute of `Ei` is a kernel function, i.e., floating point evaluation is fast.

Example 1. We demonstrate some calls with exact and symbolic input data:

```
>> Ei(1), Ei(sqrt(2)), Ei(x + 1), Ei(infinity), Ei(-infinity)

                       1/2
         Ei(1), Ei(2    ), Ei(x + 1), 0, -infinity
```

Floating point values are computed for floating point arguments:

```
>> Ei(-1000.0), Ei(1.0), Ei(12.3), Ei(2.0 + 10.0*I)

 - 1.972045137e431 - 3.141592654 I, 0.2193839344,

    0.0000003439533949, 0.003675663008 + 0.01234609005 I
```

Example 2. `Ei` is singular at the origin:

```
>> Ei(0)

 Error: singularity [Ei]
```

The negative real axis is a branch cut of `Ei`. A jump of height $2\pi i$ occurs when crossing this cut:

```
>> Ei(-1.0), Ei(-1.0 + 10^(-10)*I), Ei(-1.0 - 10^(-10)*I)

 - 1.895117816 - 3.141592654 I, - 1.895117816 - 3.141592653 I,

    - 1.895117816 + 3.141592653 I
```

Example 3. The functions `diff`, `float`, `limit`, and `series` handle expressions involving `Ei`:

```
>> diff(Ei(x), x, x, x), float(ln(3 + Ei(sqrt(PI))))

            exp(-x)   2 exp(-x)   2 exp(-x)
          - ------- - --------- - ---------, 1.120796995
               x           2           3
                          x           x

>> limit(Ei(2*x^2/(1+x)), x = infinity)

                               0

>> series(Ei(x), x = 0, 3),
   series(Ei(x), x = infinity, 3),
   series(Ei(x), x = -infinity, 3)

                          2
                         x      3
 - (ln(x) + EULER) + x - -- + O(x ),
                          4

      1          1           2         /     1     \
   -------- - --------- + --------- + O| --------- |,
   x exp(x)    2           3           |  4        |
              x  exp(x)   x  exp(x)    \ x  exp(x) /

   exp(-x)   exp(-x)   2 exp(-x)    / exp(-x) \
   ------- - ------- + --------- + O| ------- |
      x         2          3        |    4    |
               x          x         \   x     /
```

Background:

- The function `Ei(x)+ln(x)` is an entire function. `Ei` has a logarithmic singularity at the origin and a branch cut along the negative real axis. The values on the negative real axis coincide with the limit "from above":

$$Ei(x) = \lim_{\epsilon \to 0_+} Ei(x + \epsilon\, i), \quad x \text{ real}, x < 0.$$

- `Ei(x)` coincides with $Ei(1, x)$ from the following family of functions:

$$Ei(n, x) = \int_1^\infty \frac{e^{-x\,t}}{t^n}\, \mathrm{d}t.$$

  These functions are related to the incomplete gamma function `igamma` by $Ei(n, x) = x^{n-1}\, igamma(1 - n, x)$. Note that float evaluation of `igamma` is presently implemented only for real $x > 0$, whereas `Ei` can be evaluated for any complex $x \neq 0$.

- The special function $ei(x) = \int_{-\infty}^{x} e^t/t \, dt$ for *real* x (to be understood as a Cauchy Principal Value integral for $x > 0$) is related to the implemented exponential integral `Ei` by $ei(x) = -Re(Ei(-x))$, i.e.:

$$ei(x) = \begin{cases} -Ei(-x) \ , \ x < 0, \\ -Ei(-x) + i\,\pi \ , \ x > 0. \end{cases}$$

- Reference: M. Abramowitz and I. Stegun, "Handbook of Mathematical Functions", Dover Publications Inc., New York (1965).

`FAIL` – indicate a failed computation

`FAIL` is a keyword of the MuPAD language. Many functions of the library use the return value `FAIL` to indicate failed computations or non-existing elements.

Call(s):

- `FAIL`

Related Functions: `error`, `NIL`, `null`

Details:

- `FAIL` is the only element of the domain `DOM_FAIL`.
- `FAIL` is used as the return value for computations that failed. Also, requesting non-existing slots of domains or function environments yields `FAIL`. Due to this behavior, library functions can try computations without provoking errors.
- A function should return `FAIL` or an error if at least one of its inputs is `FAIL`.

Example 1. The following attempt to convert `sqrt(3)` to an integer of a residue class ring must fail:

```
>> poly(sqrt(3)*x, [x], Dom::IntegerMod(3))

                         FAIL
```

The following matrix is not invertible. You can try to invert it without producing an error:

```
>> A := matrix([[1, 1], [1, 1]]): 1/A

                         FAIL
```

The "inverse" slot of a function environment yields the inverse of the function. The inverse of the sine function is implemented, but MuPAD does not know the inverse of the dilogarithm function:

```
>> sin::inverse, dilog::inverse

                        "arcsin", FAIL

>> delete A:
```

Example 2. Most functions return FAIL or an error on input of FAIL:

```
>> poly(FAIL)

                             FAIL

>> sin(FAIL)

 Error: argument must be of 'Type::Arithmetical' [sin]
```

Example 3. FAIL evaluates to itself:

```
>> FAIL, eval(FAIL), level(FAIL, 5)

                       FAIL, FAIL, FAIL
```

FILEPATH – the pathname of a file that is currently loaded

FILEPATH is a variable containing the path of a currently read file.

Call(s):

- FILEPATH

Related Functions: fclose, fileIO, fopen, fread, package, pathname, read, READPATH

Details:

- The variable `FILEPATH` represents the pathname of a file. It only has a value while reading a file via `read` or `fread` and corresponds to the path specified in `read` or `fread`. It can only be accessed from inside the file that is currently read. Using this variable, the read file can access its own pathname and read other files via absolute pathnames, even if it only knows their relative locations with respect to itself.
- The value of `FILEPATH` is a string containing the operating system dependent path to the file that is currently read. The path string terminates with a path separator and, under Windows, starts with the name of the current volume if this was specified in the `read`/`fread` command. Cf. example 1.
- See `?fileIO` for a survey of all MuPAD functions for reading and writing files.

Example 1. Assume that the file `C:\TEMP\file.mu` contains the following lines of code. It queries its own location via `FILEPATH` (= `C:\TEMP\`) and reads two files installed relative to the location of `file.mu` via their absolute pathnames `C:\TEMP\SubFolder\file1.mu` and `C:\TEMP\SubFolder\file2.mu`, respectively:

```
print(Unquoted, "FILEPATH" = FILEPATH):
read(FILEPATH.pathname("SubFolder")."file1.mu"):
read(FILEPATH.pathname("SubFolder")."file2.mu"):
```

When reading the file `file.mu`, the part `C:\TEMP\` of the specified path is accessed by `file.mu` via `FILEPATH`. It finds the files `file1.mu` and `file2.mu` if they were installed correctly relative to the path of `file.mu`:

```
>> read("C:".pathname(Root, "TEMP", "file.mu"))

                    FILEPATH = C:\TEMP\
```

It is good programming style to use platform independent path strings. For this reason, we used the function `pathname` rather than a mere string concatenation to append appropriate path delimiters.

Changes:

- `FILEPATH` is a new variable.

`HISTORY` – the maximal number of elements in the history table

The environment variable `HISTORY` determines the maximal number of entries of the history table at interactive level.

Call(s):

- HISTORY
- HISTORY := n

Parameters:

n — a nonnegative integer smaller than 2^{31}.

Related Functions: history, last

Details:

- The commands that are entered interactively in a MuPAD session, executed in a procedure, or read from a file, as well as the resulting MuPAD outputs are stored in an internal data structure, the history table. HISTORY determines the maximal number of entries of this table at interactive level. Only the most recent entries are kept in memory.
- Entries of the history table can be accessed via history or last.
- The default value of HISTORY is 20; HISTORY has this value after starting or resetting the system via reset. Also the command delete HISTORY restores the default value.
- Within a procedure, the maximal number of entries in the local history table of the procedure is always 3, independent of the value of HISTORY.

Example 1. In the following example, we set the value of HISTORY to 2. Afterwards, only the two most recent inputs and outputs are stored in the history table at interactive level:

```
>> HISTORY := 2:
   a := 1: b := 2: max(a, b):
   history(history() - 1), history(history())

                  [(b := 2), 2], [max(a, b), 2]
```

The attempt to access the third last entry in the history table leads to an error:

```
>> history(history() - 2)

 Error: Illegal argument [history]
```

We use delete to restore the default value of HISTORY:

```
>> delete HISTORY: HISTORY
```

`LEVEL` – substitution depth of identifiers

The environment variable `LEVEL` determines the maximal substitution depth of identifiers.

Call(s):

- `LEVEL`
- `LEVEL := n`

Parameters:

`n` — a positive integer smaller than 2^{31}.

Further Documentation: Chapter 5 of the MuPAD Tutorial.

Related Functions: `context`, `eval`, `hold`, `level`, `MAXLEVEL`, `MAXDEPTH`, `val`

Details:

- When a MuPAD object is evaluated, identifiers occurring in it are replaced by their values. This happens recursively, i.e., if the values themselves contain identifiers, then these are replaced as well. `LEVEL` determines the maximal recursion depth of this process.
- Technically, evaluation of a MuPAD object works as follows. For a compound object, usually first the operands are evaluated recursively, and then the object itself is evaluated. E.g., if the object is a function call with arguments, the arguments are evaluated first, and then the function is executed with the evaluated arguments.

  With respect to the evaluation of identifiers, the *current substitution depth* is recorded internally. Initially, this value is zero. If an identifier is encountered during the recursive evaluation process as described above and the current substitution depth is smaller than `LEVEL`, then the identifier is replaced by its value, the current substitution depth is increased by one, and evaluation proceeds recursively with the value of the identifier. After the identifier has been evaluated, the current substitution depth is reset to its previous value. If the current substitution depth equals `LEVEL`, however, then the recursion stops and the identifier remains unevaluated.
- The default value of `LEVEL` at interactive level is `100`. However, the default value of `LEVEL` within a procedure is `1`. Then an identifier is only replaced by its value, which is not evaluated recursively.

The value of `LEVEL` may be changed within a procedure, but it is reset to `1` each time a new procedure is entered. After the procedure returns, `LEVEL` is reset to its previous value. See example 3.

- The evaluation of local variables and formal parameters of procedures, of type `DOM_VAR`, is not affected by `LEVEL`: they are always evaluated with substitution depth `1`. This means that a local variable or a formal parameter is replaced by its value when evaluated, but the value is not evaluated further.
  See example 3.

- `LEVEL` does not affect on the evaluation of arrays, tables and polynomials.
  See example 4.

- The function `eval` evaluates its argument with substitution depth given by `LEVEL`, and then evaluates the result again with the same substitution depth.

  The call `level(object, n)` evaluates its argument with substitution depth `n`, independent of the value of `LEVEL`.

- If, during evaluation, the substitution depth `MAXLEVEL`, is reached, then the evaluation is terminated with an error. This is a heuristic for recognizing recursive definitions, as in the example `delete a; a := a + 1; a`. Here, `a` would be replaced by `a + 1` infinitely often. Note that this has no effect if `MAXLEVEL` is greater than `LEVEL`. The default value of `MAXLEVEL` is `100`, i.e., it is equal to the default value of `LEVEL` at interactive level. However, unlike `LEVEL`, `MAXLEVEL` is not changed within a procedure, and hence recursive definitions are usually not recognized within procedures. See the help page of `MAXLEVEL` for examples.

- The default value of `LEVEL` is `100` at interactive level; `LEVEL` has this value after starting or resetting the system via `reset`. Within a procedure, the default value is `1`. The command `delete LEVEL` restores the default value.

Example 1. We demonstrate the effect of various values of `LEVEL` at interactive level:

```
>> delete a0, a1, a2, a3, a4, b: b := b + 1:
   a0 := a1: a1 := a2 + 2: a2 := a3 + a4: a3 := a4^2: a4 := 5:

>> LEVEL := 1: a0, a0 + a2, b;
   LEVEL := 2: a0, a0 + a2, b;
   LEVEL := 3: a0, a0 + a2, b;
   LEVEL := 4: a0, a0 + a2, b;
   LEVEL := 5: a0, a0 + a2, b;
   LEVEL := 6: a0, a0 + a2, b;
   delete LEVEL:
```

```
                 a1, a1 + a3 + a4, b + 1

                                 2
               a2 + 2, a2 + a4  + 7, b + 2

            a3 + a4 + 2, a3 + a4 + 32, b + 3

                  2        2
                a4  + 7, a4  + 37, b + 4

                      32, 62, b + 5

                      32, 62, b + 6
```

Example 2. In the following calls, the identifier `a` is fully evaluated:

```
>> delete a, b, c:
   a := b: b := c: c := 7: a

                                 7
```

After assigning the value `2` to `LEVEL`, `a` is evaluated only with depth two:

```
>> LEVEL := 2: a;
   delete LEVEL:

                                 c
```

If we set `MAXLEVEL` to `2` as well, evaluation of `a` produces an error, although there is no recursive definition involved:

```
>> LEVEL := 2: MAXLEVEL := 2: a

 Error: Recursive definition [See ?MAXLEVEL]

>> delete LEVEL, MAXLEVEL:
```

Example 3. This example shows the difference between the evaluation of identifiers and local variables. By default, the value of `LEVEL` is `1` within a procedure, i.e., a global identifier is replaced by its value when evaluated, but there is no further recursive evaluation. This changes when `LEVEL` is assigned a bigger value inside the procedure:

```
>> delete a0, a1, a2, a3:
   a0 := a1 + a2:  a1 := a2 + a3:  a2 := a3^2 - 1:  a3 := 5:
   p := proc()
          save LEVEL;
        begin
          print(a0, eval(a0)):
          LEVEL := 2:
          print(a0, eval(a0)):
        end_proc:
```

```
>> p()
```

```
                                        2
                  a1 + a2, a2 + a3 + a3  - 1

                                  2
                     a2 + a3 + a3  - 1, 53
```

In contrast, evaluation of a local variable replaces it by its value, without further evaluation. When `eval` is applied to an object containing a local variable, then the effect is an evaluation of the value of the local variable with substitution depth `LEVEL`:

```
>> q := proc()
          save LEVEL;
          local x;
        begin
          x := a0:
          print(x, eval(x)):
          LEVEL := 2:
          print(x, eval(x)):
        end_proc:
   q()
```

```
                                        2
                  a1 + a2, a2 + a3 + a3  - 1

                                   2
                       a1 + a2, a3  + 28
```

The command `x:=a0` assigns the value of the identifier `a0`, namely the unevaluated expression `a1+a2`, to the local variable `x`, and `x` is replaced by this value every time it is evaluated, independent of the value of `LEVEL`.

Example 4. `LEVEL` does not affect on evaluation of `polynomials`:

```
>> delete a, x:  p := poly(a*x, [x]):  a := 2:  x := 3:
   p, eval(p);
   LEVEL := 1: p, eval(p);
   delete LEVEL:
```

```
poly(a x, [x]), poly(a x, [x])

poly(a x, [x]), poly(a x, [x])
```

The same is true for `arrays` and `tables`:

```
>> delete a, b:
   A := array(1..2, [a, b]):  T := table(a = b):
   a := 1:   b := 2:
   A, eval(A), T, eval(T);
   LEVEL := 1: A, eval(A), T, eval(T);
   delete LEVEL:

              +-    -+  +-    -+  table(    table(
              | a, b |, | a, b |,   a = b ,   a = b
              +-    -+  +-    -+  )         )

              +-    -+  +-    -+  table(    table(
              | a, b |, | a, b |,   a = b ,   a = b
              +-    -+  +-    -+  )         )
```

`Line-Editor` – editing lines in the terminal version of MuPAD

This page describes the line editing facility of MuPAD's terminal version.

Details:

- The line editor described below is only available in the terminal version of MuPAD, not under the X-Window-System, nor on the Macintosh, nor under Windows.
- The current text line can be edited with the line editor during interactive input. Most of the following editor commands are entered by pressing the control key together with a second key. The available commands are:

<Ctrl-A>	– Moves the cursor to the beginning of the line.
<Ctrl-Y>	– Moves the cursor to the beginning of the previous word. This does not work under Solaris, where <Ctrl-Y> raises a non-POSIX signal which suspends the session.
<Ctrl-B>, <Cursor-Left>	– Moves the cursor one character to the left.
<Ctrl-F>, <Cursor-Right>	– Moves the cursor one character to the right.
<Ctrl-E>	– Moves the cursor to the end of the line.
<Ctrl-U>	– Deletes the complete input line.
<Ctrl-W>	– Deletes all characters from the cursor position to the beginning of the previous word.
<Ctrl-H>	– Deletes the character left of the cursor.
<Ctrl-D>	– Deletes the character at the cursor position.
<Ctrl-T>	– Deletes the next word.
<Ctrl-K>	– Deletes all characters to the end of the line.
<Ctrl-L>	– Inserts the last input line before the current cursor position.
<Ctrl-P>, <Cursor-Up>	– Reproduces the last input line. Repeated pressing of <Ctrl-P> successively reproduces the previous input lines. If the cursor is not at the beginning of the line then the previous lines are searched for an entry that corresponds to the characters of the current input.
<Ctrl-N>, <Cursor-Down>	– The analogue of <Ctrl-P>, but the previous input is run through in reverse order.
<Ctrl-C>	– Used during editing, the MuPAD input will be ignored; the MuPAD prompt appears for a new input. Used directly after the MuPAD prompt, the MuPAD process is terminated. Used during a MuPAD calculation, the computation is interrupted.
<TAB>	– Completes the actual input to the name of a system object. This may be the name of a library, of a function, or of an environment variable, respectively. If the actual input matches the beginning of several system objects, then all completed names are printed to the screen.

Example 1. We demonstrate the <TAB> completion. The <TAB> character is pressed after the input lin. The system responds by printing the three system objects beginning with lin. These are the libraries linalg, linopt, and the system function linsolve, respectively:

```
>> lin<TAB>

 linalg, linopt, linsolve
```

The following input lists all functions of the linalg library beginning with 'a':

```
>> linalg::a<TAB>
```

```
linalg::addCol, linalg::addRow, linalg::adjoint, linalg::angle
```

The following input lists all functions available in the `groebner` library:

```
>> groebner::<TAB>
```

```
groebner::dimension, groebner::gbasis, groebner::normalf,
groebner::spoly
```

`MAXDEPTH` – prevent infinite recursion during procedure calls

The environment variable `MAXDEPTH` determines the maximal recursion depth of nested procedure calls. When this recursion depth is reached, an error occurs.

Call(s):

- `MAXDEPTH`
- `MAXDEPTH:= n`

Parameters:

`n` — a positive integer smaller than 2^{31}.

Related Functions: `eval`, `freeze`, `LEVEL`, `level`, `MAXLEVEL`, `proc`

Details:

- The purpose of `MAXDEPTH` is to provide a heuristic for recognizing infinite recursion with respect to procedure calls, like in `p := x -> p(x): p(0)`. If, in this example, the recursion depth would not be limited, then the procedure `p` would call itself recursively infinitely often, and the system would "hang".
- If during the evaluation of an object the recursion depth `MAXDEPTH` is reached, then the computation is aborted with an error.
- Similarly, the environment variable `MAXLEVEL` provides a heuristic for recognizing infinite recursion with respect to the substitution of values for identifiers; see the corresponding help page for details and examples.
- The default value of `MAXDEPTH` is `500`; `MAXDEPTH` has this value after starting or resetting the system via `reset`. Also the command `delete MAXDEPTH` restores the default value.
- `MAXDEPTH` is a global variable. Use the statement `save MAXDEPTH` in a procedure to confine any changes to `MAXDEPTH` to this procedure.

Example 1. Evaluation of objects defined by an infinite recursion produces an error:

```
>> p := proc() begin p() end_proc: p()

 Error: Recursive definition [See ?MAXDEPTH];
 during evaluation of 'p'
```

This also works for mutually recursive definitions:

```
>> p := proc(x) begin q(x + 1)^2 end_proc:
   q := proc(y) begin p(x) + 2 end_proc:
   p(0)

 Error: Recursive definition [See ?MAXDEPTH];
 during evaluation of 'p'
```

Example 2. If the maximal recursion depth is reached, then this does not necessarily mean that infinite recursion is involved. The following recursive procedure computes the factorial of a nonnegative integer. If we set the maximal recursion depth to a smaller value than necessary to compute 4!, then an error occurs:

```
>> factorial := proc(n) begin
     if n = 0 then 1
     else n*factorial(n - 1)
     end_if
   end_proc:
   MAXDEPTH := 4: factorial(5)

 Error: Recursive definition [See ?MAXDEPTH];
 during evaluation of 'factorial'
```

If we set `MAXDEPTH` to 5, then the recursion depth is big enough for computing 4!. The command `delete MAXDEPTH` resets `MAXDEPTH` to its default value `500`:

```
>> MAXDEPTH := 5: factorial(5); delete MAXDEPTH:
```

120

`MAXLEVEL` – prevent infinite recursion during evaluation

The environment variable `MAXLEVEL` determines the maximal substitution depth of identifiers. When this substitution depth is reached, an error occurs.

Call(s):

- `MAXLEVEL`
- `MAXLEVEL:= n`

Parameters:

`n` — an integer between 2 and 2^{31}.

Related Functions: `context`, `eval`, `hold`, `LEVEL`, `level`, `MAXDEPTH`, `val`

Details:

- When a MuPAD object is evaluated, identifiers occurring in it are replaced by their values. This happens recursively, i.e., if the values themselves contain identifiers, then these are replaced as well. `MAXLEVEL` determines the maximal recursion depth of this process. If the substitution depth `MAXLEVEL` is reached, then an error occurs.

- The purpose of `MAXLEVEL` is to provide a heuristic for recognizing infinite recursion with respect to the replacement of identifiers by their values, like in `delete a: a := a + 1; a`. If, in this example, the substitution depth would not be limited, then `a + 1` would be substituted for `a` infinitely often, and the system would "hang".

- Similarly, the environment variable `MAXDEPTH` provides a heuristic for recognizing infinite recursion with respect to function calls; see the corresponding help page for details.

- There is a close connection between `LEVEL` and `MAXLEVEL`. If the substitution depth `LEVEL` is reached during the evaluation process, then the recursion stops and any remaining identifiers remain unevaluated, but no error occurs.

  Thus, if `MAXLEVEL > LEVEL`, then `MAXLEVEL` has no effect. By default, `LEVEL` and `MAXLEVEL` have the same value `100` at interactive level. However, the default value of `LEVEL` within a procedure is `1`, and thus usually `MAXLEVEL` has no effect within procedures.

- There are some notable differences between `LEVEL` and `MAXLEVEL`. The value of `LEVEL` depends on the context, namely whether the evaluation happens at interactive level or in a procedure. Moreover, some system functions, such as `context` and `level`, do not respect the current value of `LEVEL`. In contrast, `MAXLEVEL` is a global bound. It works as a last resort when the control of the evaluation via `LEVEL` fails.

- The default value of `MAXLEVEL` is `100`; `MAXLEVEL` has this value after starting or resetting the system via `reset`. Also the command `delete MAXLEVEL` restores the default value.

- MAXLEVEL is a global variable. Use the statement `save MAXLEVEL` in a procedure to confine any changes to MAXLEVEL to this procedure.

Example 1. Evaluation of objects defined by an infinite recursion produces an error:

```
>> delete a: a := a + 1: a

 Error: Recursive definition [See ?MAXLEVEL]
```

This also works for mutually recursive definitions:

```
>> delete a, b: a := b^2: b := a + 1: b

 Error: Recursive definition [See ?MAXLEVEL]
```

Example 2. If MAXLEVEL is smaller or equal to LEVEL, as is the default at interactive level, then objects are evaluated completely up to depth MAXLEVEL-1, and an error occurs if the substitution depth MAXLEVEL is reached, whether a recursive definition is involved or not:

```
>> delete a, b, c, d:
   a := b: b := c: c := 7: d := d + 1:
   MAXLEVEL := 2: LEVEL := 2: c

                                  7

>> a

 Error: Recursive definition [See ?MAXLEVEL]

>> d

 Error: Recursive definition [See ?MAXLEVEL]
```

On the other hand, MAXLEVEL has no effect if it exceeds LEVEL. Then any object is evaluated up to depth at most LEVEL, and the “recursive definition” error does not occur:

```
>> MAXLEVEL := 3: a, b, c, d

                           c, 7, 7, d + 2
```

In particular, MAXLEVEL normally has no effect within procedures, where by default LEVEL has the value 1:

```
>> MAXLEVEL := 2:
   p := proc() begin a, d end_proc:
   p();
   delete MAXLEVEL, LEVEL:
```

```
b, d + 1
```

NIL – the singleton element of the domain DOM_NIL

NIL is a keyword of the MuPAD language which represents the singleton element of the domain DOM_NIL.

Call(s):

- NIL

Related Functions: delete, FAIL, null

Details:

- The kernel domain DOM_NIL has only one singleton element. NIL is a keyword of the MuPAD language which represents this element. NIL is not changed by evaluation, see DOM_NIL.
- Most often, NIL is used to represent a "missing" or "void" operand in a data structure. The "void object" returned by null is not suitable for this, because it is removed from most containers (like lists, sets or expressions) during evaluation.
- When a new array from the kernel domain DOM_ARRAY is created, its elements are initialized with the value NIL. The function op returns NIL for un-initialized array elements. Note, however, that an indexed access of an un-initialized array element returns the indexed expression instead of NIL.
- Local variables of procedures defined by proc are initialized with NIL. Nevertheless, a warning is printed if one accesses a local variable without explicitly initializing its value.
- In former versions of MuPAD, NIL was used to delete values of identifiers or entries of arrays or tables, by assigning NIL to the identifier or entry. This is no longer supported. One must use delete to delete values. NIL now is a valid value of an identifier and a valid entry of an array or table.

Example 1. Unlike the "void object" returned by null, NIL is not removed from lists and sets:

```
>> [1, NIL, 2, NIL], [1, null(), 2, null()],
   {1, NIL, 2, NIL}, {1, null(), 2, null()}
```

```
[1, NIL, 2, NIL], [1, 2], {NIL, 1, 2}, {1, 2}
```

Example 2. `NIL` is used to represent "missing" entries of procedures. For example, the simplest procedure imaginable has the following operands:

```
>> op(proc() begin end)

 NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL, NIL
```

The first `NIL`, for example, represents the empty argument list, the second the void list of local variables and the third the void set of procedure options.

Example 3. Array elements are initialized with `NIL` if not defined otherwise. Note, however, that the indexed access for such elements yields the indexed expression:

```
>> A := array(1..2): A[1], op(A,1)

                              A[1], NIL

>> delete A:
```

Example 4. Local variables in procedures are implicitly initialized with `NIL`. Still, a warning is printed if one uses the variable without explicitly initializing it:

```
>> p := proc() local l; begin print(l) end:  p():

 Warning: Uninitialized variable 'l' used;
 during evaluation of 'p'

                                NIL

>> delete p:
```

Example 5. `NIL` may be assigned to an identifier or indexed identifier like any other value. Such an assignment no longer deletes the value of the identifier:

```
>> a := NIL:  b[1] := NIL:  a, b[1]

                              NIL, NIL

>> delete a, b:
```

NOTEBOOKFILE, NOTEBOOKPATH – Notebook file name and path

The environment variables NOTEBOOKFILE and NOTEBOOKPATH store the absolute file name and the directory name, respectively, of the current Notebook in MuPAD Pro for Windows as a string.

Call(s):

- NOTEBOOKFILE
- NOTEBOOKPATH

Related Functions: LIBPATH, READPATH, TESTPATH, UNIX, WRITEPATH

Details:

- The environment variable NOTEBOOKFILE stores the name of the current Notebook that is connected to the MuPAD kernel.
- The environment variable NOTEBOOKPATH stores the name of the directory where the current Notebook is located.
- These variables are useful, for example, when reading files that are located relative to the Notebook.

  Both variables only have a value if the Notebook has a name, which is generally the case when an existing Notebook has been opened or a new Notebook has been saved.
- The name given by NOTEBOOKFILE is an absolute file name.
- Both variables are read-only and are write-protected. One cannot assign a new value to NOTEBOOKFILE in order to change the name of the Notebook.
- NOTEBOOKFILE and NOTEBOOKPATH are only defined in MuPAD Pro for Windows. On other platforms, the two variables are just normal identifiers.

Example 1. In MuPAD Pro for Windows, one may supply start-up commands for a Notebook, which are executed when the Notebook is connected to a kernel. (See the menu File/Properties in the on-line help.)

In the start-up commands one may use NOTEBOOKPATH to read a source file "my_init.mu" which is stored in the directory of the Notebook:

```
>> fread(NOTEBOOKPATH."my_init.mu")
```

`O` – the domain of order terms (Landau symbols)

`O(f, x = x0)` represents the Landau symbol $O(f, x \to x_0)$.

Call(s):

- `O(f <, x = x0, y = y0, ...>)`

Parameters:

`f`	—	an arithmetical expression representing a function in `x`, `y` etc.
`x, y, ...`	—	the variables: identifiers
`x0, y0, ...`	—	the limit points: arithmetical expressions

Return Value: an element of the domain `O`.

Related Functions: `asympt`, `limit`, `series`, `taylor`

Details:

- Mathematically, for a function f in the variables $(x, y, \dots)$, the Landau symbol

  $$g := O(f, x \to x_0, y \to y_0, \dots)$$

  is a function in these variables with the following property: there exists a constant c and a neighborhood of the limit point $(x_0, y_0, \dots)$ such that $|g| \le c\,|f|$ for all values $(x, y, \dots)$ in that neighborhood.

  NOTE

  Typically, Landau symbols are used to denote the order terms ("error terms") of series expansions. Note, however, that the series expansions produced by `asympt`, `series`, and `taylor` represent order terms as a part of the data structures `Series::Puiseux` and `Series::gseries`; they do *not* use the domain `O`.

- With the equations `x = x0`, `y = y0` etc., `f` is regarded as a function of the specified variables. All other identifiers contained in `f` are regarded as constant parameters.

  If no variables and limit points are specified, then all identifiers in `f` are used as variables, each tending to the default limit point 0.

- Presently, only finite limit points are admissible.

- Variables tending to 0 are not printed on the screen.

- The variables of an order term may be obtained with the function `indets`. The limit points may be queried with the function `O::points`.

- The arithmetical operations `+`, `-`, `*`, `/`, and `^` are overloaded for order terms.
- Automatic simplifications are currently restricted to polynomial expressions `f`. Univariate polynomial expressions are reduced to the leading monomial of the expansion around the limit point. In multivariate polynomial expressions, all terms are discarded that are divisible by lower order terms. For non-polynomial expressions, only integer factors are removed.

Example 1. For polynomial expressions, certain simplifications occur:

```
>> O(x^4 + 2*x^2), O(7*x^3), O(x, x = 1)

                       2       3
                    O(x ), O(x ), O(1, x = 1)
```

A zero limit point is not printed on the screen:

```
>> O(1), O(1, x = 1), O(x^2/(y + 1), x = 0, y = -1, z = PI)

                              /    2                   \
                              |   x                    |
          O(1), O(1, x = 1), O| -----, z = PI, y = -1 |
                              \ y + 1                  /
```

The arithmetical operations are overloaded for order terms:

```
>> 7*O(x), O(x^2) + O(x^13), O(x^3) - O(x^3), O(x^2)^2 + O(x^4)

                             2       3       4
                    O(x), O(x ), O(x ), O(x )
```

Example 2. For multivariate polynomial expression, higher order terms are discarded if they are divisible by lower order terms:

```
>> O(15*x*y^2 + 3*x^2*y + x^2*y^2)

                               2    2
                        O(5 x y  + x  y)

>> O(x + x^2*y) = O(x)*O(1 + x*y)

                           O(x) = O(x)
```

Example 3. We demonstrate how to access the variables and the limit points of an order term:

```
>> a := O(x^2*y^2)

                                  2  2
                               O(x  y )

>> indets(a) = O::indets(a), O::points(a)

                 {x, y} = {x, y}, {x = 0, y = 0}

>> delete a:
```

ORDER – the default number of terms in series expansions

The environment variable `ORDER` controls the default number of terms that the system returns when you compute a series expansion.

Call(s):

- `ORDER`
- `ORDER := n`

Parameters:

`n` — a positive integer less than 2^{31}. The default value is 6.

Related Functions: `asympt`, `limit`, `O`, `series`, `taylor`

Details:

- The functions `taylor`, `series`, and `asympt` have an optional third argument specifying the desired number of terms of the requested series expansion, counting from the dominant term on (relative order). If this optional argument is missing, then the value of `ORDER` is used instead.
- `ORDER` may also affect the results returned by the function `limit`.
- Deletion via the statement "`delete ORDER`" resets `ORDER` to its default value 6. Executing the function `reset` also restores the default value.
- In some cases, the number of terms returned by `taylor`, `series`, or `asympt` may not agree with the value of `ORDER`. Cf. example 2.

Example 1. In the following example, we compute the first 6 terms of the series expansion of the function `exp(x)/x^2` around the origin:

```
>> series(exp(x)/x^2, x = 0)

                          2     3
       1   1         x   x     x       4
       -- + - + 1/2 + - + -- + --- + O(x )
        2   x         6   24   120
       x
```

To obtain the first 10 terms, we specify the third argument of `series`:

```
>> series(exp(x)/x^2, x = 0, 10)

                    2     3     4     5      6       7
  1   1         x   x     x     x     x      x       x
  -- + - + 1/2 + - + -- + --- + --- + ---- + ----- + ------ +
  2   x         6   24   120   720   5040   40320   362880
 x

      8
   O(x )
```

Alternatively, we increase the value of `ORDER`. This affects all subsequent calls to `series` or any other function returning a series expansion:

```
>> ORDER := 10: series(exp(x)/x^2, x = 0)

                    2     3     4     5      6       7
  1   1         x   x     x     x     x      x       x
  -- + - + 1/2 + - + -- + --- + --- + ---- + ----- + ------ +
  2   x         6   24   120   720   5040   40320   362880
 x

      8
   O(x )
```

```
>> taylor(x^2/(1 - x), x = 0)

    2    3    4    5    6    7    8    9    10     11       12
   x  + x  + x  + x  + x  + x  + x  + x  + x   + x   + O(x  )
```

Finally, we reset `ORDER` to its default value 6:

```
>> delete ORDER: taylor(x^2/(1 - x), x = 0)

        2    3    4    5    6    7      8
       x  + x  + x  + x  + x  + x  + O(x )
```

Example 2. The number of terms returned by `series` may differ from the value of `ORDER` when cancellation or rational exponents occur:

```
>> ORDER := 3:

>> series(exp(x) - 1 - x - x^2/2 - x^3/6, x = 0)

                        4      5
                       x      x        6
                       -- + --- + O(x )
                       24   120

>> series(1/(1 - sqrt(x)), x = 0)

                 1/2        3/2    2    5/2      3
           1 + x    + x + x    + x  + x    + O(x )

>> delete ORDER:
```

`path variables` – file search paths

`LIBPATH` determines the directories, where the functions `loadlib` and `loadproc` search for library files.

`PACKAGEPATH` determines the directories, where the function `package` searches for packages.

`READPATH` determines the directories, where the function `read` searches for files.

`WRITEPATH` determines the directory into which the functions `fopen`, `fprint`, `write`, and `protocol` write files.

Call(s):

- `LIBPATH := path`
- `PACKAGEPATH := path`
- `READPATH := path`
- `WRITEPATH := path`

Parameters:

`path` — the path name: a string or a sequence of strings.

Related Functions: `fclose`, `FILEPATH`, `finput`, `fopen`, `fprint`, `fread`, `ftextinput`, `loadlib`, `loadproc`, `NOTEBOOKFILE`, `NOTEBOOKPATH`, `package`, `pathname`, `print`, `protocol`, `read`, `TESTPATH`, `UNIX`, `write`

Details:

- `LIBPATH` determines the directories where library files are searched for by the functions `loadproc` and `loadlib`. By default, in the UNIX/Linux version of MuPAD, `LIBPATH` is the subdirectory `$MuPAD_ROOT_PATH/share/lib`. It can be re-defined by calling MuPAD with the command line option `-l`.

- `PACKAGEPATH` determines the search path for the function `package`. `package` searches for a package in the directories given by `PACKAGEPATH`.

- The default of the path variable `PACKAGEPATH` are the subdirectories `packages` of the MuPAD installation and directory `.mupad` in the users home directory. Cf. example 3.

  Additionaly paths can be given by calling MuPAD with the command line option `-p`.

- `READPATH` determines the search path for the function `read`. First, `read` searches for a file in the directories given by `READPATH`, then in the "working directory", and, finally, in the directories given by `LIBPATH`.

- The variables `LIBPATH`, `PACKAGEPATH` and `READPATH` can represent more than one search directory. These variables can be assigned a sequence of strings: each element of the sequence represents a directory in which files are search for.

- `WRITEPATH` determines the directory, into which the functions `fopen`, `fprint`, `write`, and `protocol` write files which are not specified with a full (absolute) pathname. If `WRITEPATH` is not defined, then the files are written into the "working directory".

- Note that the "working directory" depends on the operating system. On Windows systems, it is the folder, where MuPAD is installed. On UNIX or Linux systems, the "working directory" is the directory where MuPAD was started.

- When concatenated with a file name, the directories given by the path variables must produce valid path names. NOTE

- Path names are system dependent. Under UNIX/Linux, a subdirectory is started with a `/`, with `:` on the Macintosh, and with a single backslash \ on Windows.

- Note that in MuPAD, a single backslash inside a character string is created by two backslashes. E.g., the MuPAD string representing the path "`C:\Programs\MuPAD`" must be defined by `"C:\\Programs\\MuPAD"`.

- The function `pathname` allows to create path names independent of the current operating system.

- Changing LIBPATH is useful for library development. You may create a sub-directory of your home directory with the same structure as the library installation tree and store modified library files there. If you prepend the name of this sub-directory to the variable LIBPATH in your startup file userinit.mu, then MuPAD first looks for library files in your local directory before searching the system directory. Cf. example 4.

Example 1. This example shows how to define a READPATH. More than one path may be given. read will look for files to be opened in the directories given by READPATH. The following produces a valid READPATH for UNIX/Linux systems only, since the path separators are hard coded in the strings:

```
>> READPATH := "math/lib/", "math/local/"

                   "math/lib/", "math/local/"
```

It is good programming style to use platform independent path strings. This can be achieved with the function pathname:

```
>> READPATH := pathname("math", "lib"),
               pathname("math", "local")

                   "math/lib/", "math/local/"
```

All path variables can be set to their default values by deleting them:

```
>> delete READPATH:
```

Example 2. The path variable WRITEPATH only accepts one path string:

```
>> WRITEPATH := "math/lib/", "math/local/"

 Error: Illegal argument [WRITEPATH]
```

Example 3. The default of the path variable PACKAGEPATH are the subdirectories packages of the MuPAD installation and directory .mupad in the users home directory:

```
>> PACKAGEPATH

   "<YourMuPADpath>/packages/", "/home/user/.mupad/packages/"
```

Example 4. Be careful when changing the `LIBPATH`. You can corrupt your MuPAD session:

```
>> LIBPATH := "does/not/exist":
   linalg::det

 Error: can't read file 'LIBFILES/linalg.mu' [loadproc]
```

You can always restore the standard search path by deleting `LIBPATH`:

```
>> delete LIBPATH:
   linalg::det

                   proc linalg::det(A) ... end
```

Changing the `LIBPATH` is useful for library development. You can build a directory `"mylib"` with the same directory structure as the MuPAD library. Let us assume that you have a patched version of the function `linalg::det` in the file `"mylib/LINALG/det.mu"`. MuPAD will try to read the file `"LINALG/det.mu"` when the function `linalg::det` is called for the first time. Since the directory `"mylib"` contains this file, it will be read instead of the corresponding file in the standard library:

```
>> reset():  Pref::verboseRead(2):
   LIBPATH := pathname("mylib"), LIBPATH:
   linalg::det

 loading package 'linalg' [<YourMuPADpath>/share/lib/]
 reading file mylib/LINALG/det.mu

                   proc linalg::det(A) ... end
```

Please restore your session:

```
>> delete LIBPATH:  Pref::verboseRead(0):
```

Changes:

- `PACKAGEPATH` was introduced.

`PRETTYPRINT` – control the formatting of output

The environment variable `PRETTYPRINT` determines whether MuPAD's results are printed in the one-dimensional or the two-dimensional format.

Call(s):

- PRETTYPRINT
- PRETTYPRINT := value

Parameters:

value — either TRUE or FALSE

Related Functions: print, TEXTWIDTH

Details:

- PRETTYPRINT controls the pretty printer, which is responsible for formatted output. If PRETTYPRINT has the value TRUE, then pretty printing is enabled for output.
- The default value of PRETTYPRINT is TRUE; PRETTYPRINT has this value after starting or resetting the system via reset. Also the command delete PRETTYPRINT restores the default value.
- On Windows platforms, PRETTYPRINT normally has no effect when "typesetting" is activated. An exception occurs for very wide MuPAD output, where PRETTYPRINT determines the output style even if the typesetting is activated.

  Typesetting is activated by default. It can be switched on or off by choosing "Options" from the "View" menu of the MuPAD main window and then clicking on "Typeset output expressions".

Example 1. The following command disables pretty printing:

```
>> PRETTYPRINT := FALSE

 FALSE
```

Now MuPAD results are printed in a one-dimensional, linearized form:

```
>> series(sin(x), x = 0, 14)

 x - 1/6*x^3 + 1/120*x^5 - 1/5040*x^7 + 1/362880*x^9 - 1/399168\
 00*x^11 + 1/6227020800*x^13 + O(x^15)
```

After setting PRETTYPRINT to TRUE again, the usual two-dimensional output format is used:

```
>> PRETTYPRINT := TRUE: series(sin(x), x = 0, 14)
```

```
      3     5     7       9         11          13
     x     x     x       x         x           x           15
 x - -- + --- - ---- + ------ - -------- + ---------- + O(x   )
      6   120   5040   362880   39916800   6227020800
```

Re, Im – real and imaginary part of an arithmetical expression

`Re(z)` returns the real part of z.

`Im(z)` returns the imaginary part of z.

Call(s):

- `Re(z)`
- `Im(z)`

Parameters:

`z` — an arithmetical expression

Return Value: an arithmetical expression.

Overloadable by: `z`

Side Effects: These functions are sensitive to properties of identifiers set via `assume`. See example 2.

Related Functions: `abs`, `assume`, `conjugate`, `rectform`, `sign`

Details:

- The intended use of `Re` and `Im` is for constant arithmetical expressions. Especially for numbers, of type `DOM_INT`, `DOM_RAT`, `DOM_FLOAT`, or `DOM_COMPLEX`, the real and the imaginary part is computed directly and very efficiently.
- `Re` and `Im` can handle symbolic expressions. Properties of identifiers are taken into account (see `assume`). An identifier without any property is assumed to be complex. See example 2.

  However, for arbitrary symbolic expressions, `Re` or `Im` may be unable to extract the real or the imaginary part of `z`, respectively. You may then use the function `rectform` (see example 3). Note, however, that using `rectform` is computationally expensive.
- If `Re` cannot extract the whole real part of `z`, then the returned expression contains symbolic `Re` and `Im` calls. The same is true for `Im`. See example 2.

Example 1. The real and the imaginary part of $2e^{1+i}$ are:

```
>> Re(2*exp(1 + I)), Im(2*exp(1 + I))

                2 cos(1) exp(1), 2 sin(1) exp(1)
```

Example 2. `Re` and `Im` are not able to extract the whole real and imaginary part, respectively, of symbolic expressions containing identifiers without a value. However, in some cases they can still simplify the input expression, as in the following two examples:

```
>> delete u, v: Re(u + v*I), Im(u + v*I)

                  Re(u) - Im(v), Im(u) + Re(v)

>> delete z: Re(z + 2), Im(z + 2)

                        Re(z) + 2, Im(z)
```

By default, identifiers without a value are assumed to represent arbitrary complex numbers. You can use `assume` to change this. The following command tells the system that `z` represents only real numbers:

```
>> assume(z, Type::Real): Re(z + 2), Im(z + 2)

                            z + 2, 0
```

Example 3. Here is another example of a symbolic expression for which `Re` and `Im` fail to extract its real and imaginary part, respectively:

```
>> delete z: Re(exp(I*sin(z))), Im(exp(I*sin(z)))

              Re(exp(I sin(z))), Im(exp(I sin(z)))
```

You may use the function `rectform`, which splits a complex expression z into its real and imaginary part and is more powerful than `Re` and `Im`:

```
>> r := rectform(exp(I*sin(z)))

 cos(sin(Re(z)) cosh(Im(z))) exp(-cos(Re(z)) sinh(Im(z))) +

    (sin(sin(Re(z)) cosh(Im(z))) exp(-cos(Re(z)) sinh(Im(z)))) I
```

Then `Re(r)` and `Im(r)` give the real and the imaginary part of `r`, respectively:

```
>> Re(r)

    cos(sin(Re(z)) cosh(Im(z))) exp(-cos(Re(z)) sinh(Im(z)))

>> Im(r)

    sin(sin(Re(z)) cosh(Im(z))) exp(-cos(Re(z)) sinh(Im(z)))
```

Example 4. Symbolic expressions of type "Re" and "Im" always have the property Type::Real, even if no identifier of the symbolic expression has a property:

```
>> is(Re(sin(2*x)), Type::Real)
```

TRUE

Example 5. Advanced users can extend the functions Re and Im to their own special mathematical functions (see section "Backgrounds" below). To this end, embed your mathematical function into a function environment f and implement the behavior of the functions Re and Im for this function as the slots "Re" and "Im" of the function environment.

If a subexpression of the form f(u,..) occurs in z, then Re and Im issue the call f::Re(u,..) and f::Im(u,..), respectively, to the slot routine to determine the real and the imaginary part of f(u,..), respectively.

For illustration, we show how this works for the sine function and the slot "Re". Of course, the function environment sin already has a "Re" slot. We call our function environment Sin in order not to overwrite the existing system function sin:

```
>> Sin := funcenv(Sin):
   Sin::Re := proc(u) // compute Re(Sin(u))
     local r, s;
   begin
     r := Re(u);
     if r = u then
       return(Sin(u))
     elif not has(r, {hold(Im), hold(Re)}) then
       s := Im(u);
       if not has(s, {hold(Im), hold(Re)}) then
         return(Sin(r)*cosh(s))
       end_if
     end_if;
     return(FAIL)
   end:
```

```
>> Re(Sin(2)), Re(Sin(2 + 3*I))
```

Sin(2), Sin(2) cosh(3)

The return value FAIL tells the function Re that Sin::Re was unable to determine the real part of the input expression. The result is then a symbolic Re call:

```
>> delete f, z: Re(2 + Sin(f(z)))
```

Re(Sin(f(z))) + 2

Background:

- If a subexpression of the form `f(u,..)` occurs in `z` and `f` is a function environment, then `Re` attempts to call the slot `"Re"` of `f` to determine the real part of `f(u,..)`. In this way, you can extend the functionality of `Re` to your own special mathematical functions.

  The slot `"Re"` is called with the arguments `u,..` of `f`. If the slot routine `f::Re` is not able to determine the real part of `f(u,..)`, then it must return `FAIL`.

  If `f` does not have a slot `"Re"`, or if the slot routine `f::Re` returns `FAIL`, then `f(u,..)` is replaced by the symbolic call `Re(f(u...))` in the returned expression.

  The same holds for the function `Im`, which attempts to call the corresponding slot `"Im"` of `f`.

  See example 5.

- Similarly, if an element `d` of a library domain `T` occurs as a subexpression of `z`, then `Re` attempts to call the slot `"Re"` of that domain with `d` as argument to compute the real part of `d`.

  If the slot routine `T::Re` is not able to determine the real part of `d`, then it must return `FAIL`.

  If `T` does not have a slot `"Re"`, or if the slot routine `T::Re` returns `FAIL`, then `d` is replaced by the symbolic call `Re(d)` in the returned expression.

  The same holds for the function `Im`, which attempts to call the corresponding slot `"Im"` of the `T`.

`RootOf` – the set of roots of a polynomial

`RootOf(f, x)` represents the symbolic set of roots of the polynomial $f(x)$ with respect to the indeterminate x.

Call(s):

- `RootOf(f, x)`
- `RootOf(f)`

Parameters:

`f` — a polynomial, an arithmetical expression representing a polynomial in `x`, or a polynomial equation in `x`
`x` — the indeterminate: typically, an identifier or indexed identifier

Return Value: a symbolic `RootOf` call, i.e., an expression of type `"RootOf"`.

Related Functions: `numeric::polyroots`, `poly`, `solve`

Details:

- `RootOf` serves as a symbolic representation of the zero set of a polynomial. Since it is generally impossible to represent the roots of a polynomial in terms of radicals, `RootOf` is often the only possible way to represent the roots symbolically. `RootOf` mainly occurs in the output of `solve` or related functions; see example 3.
- The parameter `f` must be either a polynomial, or an arithmetical expression representing a polynomial in `x`, or an equation `p=q`, where `p` and `q` are arithmetical expressions representing polynomials in `x`. In the latter case, `RootOf` represents the roots of `p-q` with respect to `x`.
- The polynomial `f` need not be irreducible or even square-free. Even if `f` has multiple roots, `RootOf` represents each of the roots only with multiplicity one.
- If `x` is omitted, then `f` must be an arithmetical expression or polynomial equation containing exactly one indeterminate, and `RootOf` represents the roots with respect to this indeterminate.
- `x` need not be an identifier or indexed identifier: it may be any expression that is neither rational nor constant.
- If `f` contains only one indeterminate, then you can apply `float` to the `RootOf` object to obtain a set of floating-point approximations for all roots; see example 3.

Example 1. Each of the following calls represents the roots of the polynomial $x^3 - x^2$ with respect to x, i.e., the set $\{0, 1\}$:

```
>> RootOf(x^3 - x^2, x), RootOf(x^3 = x^2, x)

                  3    2                3    2
         RootOf(x  - x , x), RootOf(x  - x , x)

>> RootOf(x^3 - x^2), RootOf(x^3 = x^2)

                  3    2                3    2
         RootOf(x  - x , x), RootOf(x  - x , x)

>> RootOf(poly(x^3 - x^2, [x]), x)

                            3    2
                   RootOf(x  - x , x)
```

In general, however, `RootOf` is only used when no explicit symbolic representation of the roots is possible.

Example 2. The first argument of `RootOf` may contain parameters:

```
>> RootOf(y*x^2 - x + y^2, x)

                            2         2
                     RootOf(y  - x + x  y, x)
```

The set of roots of a polynomial is treated like an expression. For example, it may be differentiated with respect to a free parameter. The result is the set of derivatives of the roots; it is expressed in terms of `RootOf`, by giving a minimal polynomial:

```
>> diff(%, y)

                         4         3    2  2      2  5
       RootOf(2 y - x + y  + 4 x y  - x  y  + 4 x  y , x)
```

For reducible polynomials, the result may be a multiple of the correct minimal polynomial.

Example 3. `solve` returns `RootOf` objects when the roots of a polynomial cannot be expressed in terms of radicals:

```
>> solve(x^5 + x + 7, x)

                                  5
                     RootOf(X1 + X1  + 7, X1)
```

You can apply the function `float` to obtain floating-point approximations of all roots:

```
>> float(%)

 {- 0.508469409 + 1.368616488 I,

    - 0.5084694089 - 1.368616488 I,

    1.213876334 + 0.9241881109 I, 1.213876334 - 0.9241881108 I,

    -1.410813851}
```

Example 4. The function `sum` is able to compute sums over all roots of a given polynomial:

```
>> sum(i^2, i = RootOf(x^3 + a*x^2 + b*x + c, x))

                              2
                             a  - 2 b
```

```
>> sum(1/(z + i), i = RootOf(x^4 - y*x + 1, x))

                      3
             y + 4 z
           ------------
                    4
           y z + z  + 1
```

Si – the sine integral function

Si(x) represents the sine integral $\int_0^x \sin(t)/t \, \mathrm{d}t$.

Call(s):

- Si(x)

Parameters:

x — an arithmetical expression

Return Value: an arithmetical expression.

Overloadable by: x

Side Effects: When called with a floating point argument, the function is sensitive to the environment variable DIGITS which determines the numerical working precision.

Related Functions: Ci, Ei, int, sin

Details:

- If x is a floating point number, then Si(x) returns the numerical value of the sine integral. The special values $Si(0) = 0$ and $Si(\pm\infty) = \pm\pi/2$ are implemented. For all other arguments, Si returns a symbolic function call.
- The reflection rule $Si(x) = -Si(-x)$ is used if the argument is a negative integer or a negative rational number. It is also used if the argument is a symbolic product involving such a factor. Cf. example 2.
- The float attribute of Si is a kernel function, i.e., floating point evaluation is fast.

Example 1. We demonstrate some calls with exact and symbolic input data:

```
>> Si(0), Si(1), Si(sqrt(2)), Si(x + 1), Si(infinity)

                             1/2               PI
              0, Si(1), Si(2   ), Si(x + 1), --
                                                2
```

Floating point values are computed for floating point arguments:

```
>> Si(-5.0), Si(1.0), Si(2.0 + 10.0*I)

     -1.549931245, 0.9460830704, 1187.409493 - 242.5252717 I
```

Example 2. The reflection rule $Si(-x) = -Si(x)$ is implemented for negative real numbers and products involving such numbers:

```
>> Si(-3), Si(-3/7), Si(-sqrt(2)), Si(-x/7), Si(-0.3*x)

                                1/2       / x \
       -Si(3), -Si(3/7), - Si(2   ), - Si| - |, -Si(0.3 x)
                                          \ 7 /
```

No such "normalization" occurs for complex numbers or arguments that are not products:

```
>> Si(- 3 - I), Si(3 + I), Si(x - 1), Si(1 - x)

          Si(- 3 - I), Si(3 + I), Si(x - 1), Si(1 - x)
```

Example 3. The functions `diff`, `float`, `limit`, and `series` handle expressions involving `Si`:

```
>> diff(Si(x), x, x, x), float(ln(3 + Si(sqrt(PI))))

            2 sin(x)   sin(x)   2 cos(x)
            -------- - ------ - --------, 1.502020149
               3         x         2
              x                   x

>> limit(Si(2*x^2/(1+x)), x = infinity)

                             PI
                             --
                              2

>> series(Si(x), x = 0), series(Si(x), x = infinity, 3)
```

```
     3     5
    x     x       7   PI   cos(x)   sin(x)    /  1 \
x - -- + --- + O(x ), -- - ------ - ------ + O| -- |
    18   600          2      x        2       |  3 |
                                     x        \ x  /
```

Background:

- Si is an entire function.
- Reference: M. Abramowitz and I. Stegun, "Handbook of Mathematical Functions", Dover Publications Inc., New York (1965).

`TESTPATH` – write path for `prog::test`

`TESTPATH` determines the directory into which the function `prog::test` writes its files.

Call(s):

- `TESTPATH := path`

Parameters:

`path` — a valid directory path: a string.

Related Functions: `LIBPATH`, `NOTEBOOKFILE`, `NOTEBOOKPATH`, `prog::test`, `READPATH`, `UNIX`, `WRITEPATH`

Details:

- `TESTPATH` is a special write path for result files generated by `prog::test`.
- The help page on ■`path variables"'` explains how to define path variables correctly and in a system independent way.

Example 1. A path name must end with a directory separator. Here is an example for UNIX platforms:

```
>> TESTPATH := "testresults/"

                    "testresults/"
```

`TEXTWIDTH` – the maximum number of characters in an output line

The environment variable `TEXTWIDTH` determines the maximum number of characters in one line of screen output.

Call(s):

- `TEXTWIDTH`
- `TEXTWIDTH := n`

Parameters:

`n` — a positive integer smaller than 2^{31}. The default value is 75.

Related Functions: `fprint`, `PRETTYPRINT`, `print`

Details:

- Output is broken into several lines if it needs more than `TEXTWIDTH` characters per line.
- Deletion via the statement "`delete TEXTWIDTH`" resets `TEXTWIDTH` to its default value. Executing the function `reset` also restores the default value.
- The minimal value of `TEXTWIDTH` depends on the length of the prompt string, which is defined via `Pref::promptString`: The minimal value is 7 plus the length of the prompt string. The default prompt string is `">> "`, thus the minimal value of `TEXTWIDTH` is 10 in this case.
- `TEXTWIDTH` is set to its maximum value $2^{31} - 1$ when printing to a text file using `fprint`. Thus, no additional line breaks occur in the output.
- `TEXTWIDTH` does not influence the typesetting of expressions which is available for some user interfaces of MuPAD.
- Most examples in this manual are printed with `TEXTWIDTH` set to 63.

Example 1. The maximal length of a line is set to 20 characters:

```
>> oldTEXTWIDTH := TEXTWIDTH:
   TEXTWIDTH := 20: 30!

 2652528598121910586\
 36308480000000
```

We restore the previous value:

```
>> TEXTWIDTH := oldTEXTWIDTH: 30!

                265252859812191058636308480000000
```

Example 2. The following procedure adds empty characters to produce output that is flushed right:

```
>> myprint := proc(x) local l; begin
                if domtype(x) <> DOM_STRING then
                   x := expr2text(x);
                end_if;
                l := length(x);
                print(Unquoted, _concat(" " $ TEXTWIDTH - l, x))
              end_proc:

>> myprint("hello world"):  myprint(30!):  myprint("bye bye"):

                                                      hello world

                                265252859812191058636308480000000

                                                          bye bye

>> delete myprint:
```

`TRUE`, `FALSE`, `UNKNOWN` – Boolean constants

MuPAD uses a three state logic with the Boolean constants `TRUE`, `FALSE`, and `UNKNOWN`.

Related Functions: `_lazy_and`, `_lazy_or`, `and`, `bool`, `DOM_BOOL`, `if`, `is`, `not`, `or`, `repeat`, `while`

Details:

- The Boolean constants `TRUE`, `FALSE`, `UNKNOWN` are of domain type `DOM_BOOL`.
- See `and`, `or`, `not` for the logical rules of MuPAD's three state logic.
- Boolean constants are returned by system functions such as `bool` and `is`. These functions evaluate Boolean expressions such as equations and inequalities.

Example 1. The Boolean constants may be combined via `and`, `or`, and `not`:

```
>> (TRUE and (not FALSE)) or UNKNOWN
```

```
TRUE
```

Example 2. The function `bool` serves for reducing Boolean expressions such as equations or inequalities to one of the Boolean constants:

```
>> bool(x = x and 2 < 3 and 3 <> 4 or UNKNOWN)
```

```
TRUE
```

The function `is` evaluates symbolic Boolean expressions with properties:

```
>> assume(x > 2): is(x^2 > 4), is(x^3 < 0), is(x^4 > 17)
```

```
TRUE, FALSE, UNKNOWN
```

```
>> unassume(x):
```

Example 3. Boolean constants occur in the conditional part of program control structures such as `if`, `repeat`, or `while` statements. The following loop searches for the smallest Mersenne prime larger than 500 (see `numlib::mersenne` for details). The function `isprime` returns `TRUE` if its argument is a prime, and `FALSE` otherwise. Once a Mersenne prime is found, the `while`-loop is interrupted by the `break` statement:

```
>> p := 500:
   while TRUE do
     p := nextprime(p + 1):
     if isprime(2^p - 1) then
        print(p);
        break;
     end_if;
   end_while:
```

```
521
```

Note that the conditional part of `if`, `repeat`, and `while` statements must evaluate to `TRUE` or `FALSE`. Any other value leads to an error:

```
>> if UNKNOWN then "true" else "false" end_if
```

```
 Error: Can't evaluate to boolean [if]
```

```
>> delete p:
```

`UNIX` – **MuPAD** command line options and initialization files for UNIX

This help page describes all command line options and initialization files for MuPAD on UNIX platforms.

Call(s):

- `mupad [-f] [-F] [-g] [-r] [-S] [-v] [-V] [-a stacksize] [-h helppath]`
  `[-l libpath] [-L primelimit] [-m modpath] [-p packagepath]`
  `[-P [pPeEsSwW]] [-u userpath] [-U opts] [-w sec] [file...]`
- `xmupad [-f] [-F] [-r] [-S] [-v] [-V] [-a stacksize] [-h helppath]`
  `[-l libpath] [-L primelimit] [-m modpath] [-p packagepath ]`
  `[-P [pPeEsSwW]] [-u userpath] [-U opts] [-w sec] [file...]`

Related Functions: `LIBPATH`, `NOTEBOOKFILE`, `NOTEBOOKPATH`, `PACKAGEPATH`, `READPATH`, `TESTPATH`, `UNIX`, `WRITEPATH`

Details:

- The following table lists all command line options of the terminal version `mupad` and the graphical user interface `xmupad`.

  The option `-g` cannot be set for the X11 front-end, where it is reserved for specifying the window geometry. However, the debug mode can be switched on in the "Options" menu of the graphical user interface.

option	description	
`-a`	size of the `PARI` arithmetic stack in bytes	(default: 250 000)
`-f`	do not read the user's initialization file	
`-F`	do not include `~/.mupad/packages` in the `PACKAGEPATH`	
`-g`	debug mode	
`-h`	path name for the help index	(default: `$R/share/doc/ascii`)

`-l`	default library path; can be changed interactively via `LIBPATH`	(default: `$R/share/lib`)
`-L`	pre-compute a list of all primes up to `primelimit`	(default: 1 000 000)
`-m`	path name for dynamic modules	(default: `$R/$A/modules`)
`-p`	additional `package path` (can be used more than once)	
`-P`	suppress (`p`) or print (`P`) prompt; can be changed interactively via `Pref::prompt`	(default: `p`)
	suppress (`e`) or echo (`E`) input; can be changed interactively via `Pref::echo`	(default: `e`)
	suppress (`w`) or print (`W`) warnings about changes in the new version of MuPAD; can be changed interactively via `Pref::warnChanges`	(default: `w`)
	start kernel in a more secure mode (`S`) or not (`s`); secure mode restricts file access and forbids the use of the MuPAD command `system`	(default: `s`)
`-r`	prints the path of the MuPAD installation directory	
`-S`	start without printing the MuPAD logo	
`-u`	path name of the user initialization file	(default: `~/.mupad/`)
`-U`	pass arbitrary options to the MuPAD session, which can be queried interactively via `Pref::userOptions`	(default: ””)
`-v`	verbose debug mode	
`-V`	prints the MuPAD version	
`-w`	terminate MuPAD process after at most `sec` seconds	

`$R` denotes the MuPAD installation directory. `$A` denotes the architecture name returned by the shell script `$R/share/bin/sysinfo`.

The locations of MuPAD's initialization files are:

`~/.mupad/userinit.mu`	user initialization file
`~/.mupad/mxdviRecentFiles`	list of recent documents (help tool)
`~/.mupad/mxmupadrc`	preferences of the X11 front-end
`~/.mupad/vcam_defaults`	defaults of the graphics renderer
`$R/share/lib/sysinit.mu`	system initialization file
`$R/share/lib/.MMMinit`	MAMMUT initialization file (memory management)

In addition to the options one or more MuPAD source files `file...` can be given on the command line. They are read in and executed in the given order.

Example 1. The following command starts the terminal version of MuPAD, which does not read the user's initialization file (`-f`) and does not display a banner (`-S`), pre-computes and stores all primes up to 2 000 000 (`-L`), and expects to find dynamic modules in the directory `myModules` (`-m`):

```
# mupad -f -S -L 2000000 -m myModules
>>
```

Changes:

- The former option `-n` was renamed to `-f`.
- The former option `-p` was renamed to `-a`.
- The package path is now set with the option `-p`.
- The new option `-F` was introduced.
- The new option `-r` was introduced.
- The new option `-V` was introduced.

`abs` – the absolute value of a real or complex number

`abs(z)` returns the absolute value of the number `z`.

Call(s):

- `abs(z)`

Parameters:

`z` — an arithmetical expression

Return Value: an arithmetical expression.

Overloadable by: `z`

Side Effects: `abs` respects properties of identifiers.

Related Functions: `conjugate`, `Im`, `norm`, `Re`, `sign`

Details:

- For many constant expressions, `abs` returns the absolute value as an explicit number or expression. Cf. example 1.
- A symbolic call of `abs` is returned if the absolute value cannot be determined (e.g., because the argument involves identifiers). The result is subject to certain simplifications. In particular, `abs` extracts constant factors. Properties of identifiers are taken into account. Cf. examples 2 and 3.
- The `expand` function rewrites the absolute value of a product to a product of absolute values. E.g., `expand(abs(x*y))` yields `abs(x)*abs(y)`. Cf. example 4.
- The symbolic constants `CATALAN`, `E`, `EULER`, and `PI` are processed by `abs`. Cf. example 5.
- The absolute value of symbolic function calls can be defined via the slot `"abs"` of function environments. Cf. example 7.
- In the same way, the absolute value of domain elements can be defined via overloading. Cf. example 8.

Example 1. For many constant expressions, the absolute value can be computed explicitly:

```
>> abs(1.2), abs(-8/3), abs(3 + I), abs(sqrt(-3))

                                     1/2   1/2
                        1.2, 8/3, 10    , 3

>> abs(sin(42)), abs(PI^2 - 10), abs(exp(3) - tan(157/100))

                                2
              -sin(42), 10 - PI , tan(157/100) - exp(3)

>> abs(exp(3 + I) - sqrt(2))

                2       2                       1/2 2 1/2
         (sin(1)  exp(3)  + (cos(1) exp(3) - 2   ) )
```

Example 2. Symbolic calls are returned if the argument contains identifiers without properties:

```
>> abs(x), abs(x + 1), abs(sin(x + y))

                abs(x), abs(x + 1), abs(sin(x + y))
```

The result is subject to some simplifications. In particular, `abs` splits off constant factors in products:

```
>> abs(PI*x*y), abs((1 + I)*x), abs(sin(4)*(x + sqrt(3)))

                                1/2                     1/2
        PI abs(x y), abs(x) 2   , - sin(4) abs(x + 3   )
```

Example 3. `abs` is sensitive to properties of identifiers:

```
>> assume(x < 0):  abs(3*x), abs(PI - x), abs(I*x)

                         -3 x, PI - x, -x

>> unassume(x):
```

Example 4. The `expand` function produces products of `abs` calls:

```
>> abs(x*(y + 1)), expand(abs(x*(y + 1)))

                abs(x (y + 1)), abs(x) abs(y + 1)
```

Example 5. The absolut value of the symbolic constants `PI`, `EULER` etc. are known:

```
>> abs(PI), abs(EULER + CATALAN^2)

                                                2
                          PI, EULER + CATALAN
```

Example 6. Expressions containing `abs` can be differentiated:

```
>> diff(abs(x), x),  diff(abs(x), x, x)

                           sign(x), 2 dirac(x)
```

Example 7. The slot `"abs"` of a function environment `f` defines the absolute value of symbolic calls of `f`:

```
>> abs(f(x))

                                abs(f(x))

>> f := funcenv(f): f::abs := x -> f(x)/sign(f(x)): abs(f(x))

                                  f(x)
                               ----------
                               sign(f(x))

>> delete f:
```

Example 8. The slot `"abs"` of a domain `d` defines the absolute value of its elements:

```
>> d := newDomain("d"):  e1 := new(d, 2): e2 := new(d, x):
   d::abs := x -> abs(extop(x, 1)):  abs(e1), abs(e2)

                                2, abs(x)

>> delete d, e1, e2:
```

`alias`, `unalias` – defines or un-defines an abbreviation or a macro

`alias(x = object)` defines `x` as an abbreviation for `object`.

`alias(f(y1, y2, ...) = object)` defines `f` to be a macro. For arbitrary objects `a1, a2, ...`, `f(a1, a2, ...)` is equivalent to `object` with `a1` substituted for `y1`, `a2` substituted for `y2`, etc.

`alias()` displays a list of all currently defined aliases and macros.

`unalias(x)` deletes the abbreviation or the macro `x`.

`unalias()` deletes all abbreviations and macros.

Call(s):

- `alias(x1 = object1, x2 = object2, ...)`
- `alias()`
- `unalias(z1, z2, ...)`
- `unalias()`

Parameters:

`x1, x2, ...`	—	identifiers or symbolic expressions of the form `f(y1, y2, ...)`, with identifiers `f`, `y1, y2, ...`
`object1, object2, ...`	—	any MuPAD objects
`z1, z2, ...`	—	identifiers

Return Value: Both `alias` and `unalias` return the void object of type `DOM_NULL`.

When called with no arguments, `alias` displays all currently defined aliases as a sequence of equations; see below for a description.

Side Effects: `alias` with at least one argument and `unalias` change the parser configuration in the way described in the "Details" section.

Related Functions: `:=`, `finput`, `fprint`, `fread`, `input`, `Pref::alias`, `print`, `proc`, `read`, `subs`, `text2expr`, `write`

Details:

- `alias(x = object)` defines an abbreviation. It changes the configuration of the parser such that the identifier `x` is replaced by `object` whenever it occurs in the input, and such that `object` is in turn replaced by `x` in the output.

- `alias(f(y1, y2, ...) = object)` defines a macro. It changes the configuration of the parser such that a function call of the form `f(a1, a2, ...)`, where `a1,a2,...` is a sequence of arbitrary objects of the same length as `y1,y2,...`, is replaced by `object` with `a1` substituted for `y1`, `a2` substituted for `y2`, etc.

No substitution takes place if the number of parameters `y1,y2,...` differs from the number of arguments `a1,a2,...`. No substitution takes place in the output.

It is valid to define a macro with no arguments via `alias(f()=object)`.

- Multiple alias definitions may be given in a single call; abbreviations and macros may be mixed.

- `alias()` displays all currently defined aliases as a sequence of equations. For an abbreviation defined via `alias(x = object)`, the equation `x = object` is printed. For a macro defined via `alias(f(y1, y2, ...) = object)`, the equation `f = [object, [y1, y2, ...]]` is printed. If no aliases are defined, the message "No alias defined" is printed. See Example 11.

- `unalias(x)` deletes the abbreviation or macro `x`. To delete a macro defined by `alias(f(y1, y2, ...) = object)`, use `unalias(f)`. If no alias for `x` or `f`, respectively, is defined currently, the call is ignored.

  Multiple alias definitions may be deleted by a single call of `unalias`. The call `unalias()` deletes all currently defined aliases.

- Neither `alias` nor `unalias` evaluate its arguments. Hence it has no effect if the aliased identifier has a value, and `alias` creates an alias for the right hand side of the alias definition and not for its evaluation. Cf. example 2.

- `alias` does not flatten its arguments. Thus an expression sequence is a valid right hand side of an alias definition. See example 5.

- An alias definition causes a substitution similar to the effect of `subs`, not just a textual replacement. Cf. example 3.

- Each identifier may be aliased to only one object. Each object may be abbreviated in only one way; otherwise `alias` aborts with an error.

- An alias is in effect from the time when the call to `alias` has been evaluated. It affects exactly those inputs that are *parsed* after that moment. Cf. example 9. In particular, an alias definition inside a procedure does not affect the rest of the procedure.

- By default, back-substitution of aliases in the output happens only for abbreviations and not for macros. After a command of the form `alias(x = object)`, both the unevaluated object `object` and its evaluation are replaced by the unevaluated identifier `x` in the output. Cf. example 2.

  The user can control the behavior of the back-substitution in the output with the function `Pref::alias`; see the corresponding help page for details.

- Substitutions in the output only happen for the results of computations at interactive level. The behavior of the functions `fprint`, `print`, or `write` is not affected.

- Alias substitutions are performed in parallel, both in the input and in the output. Thus it is not possible to define nested aliases. See Example 10.
- If an identifier is used as an abbreviation, it is not possible to enter this identifier in its literal meaning any longer.

  In particular, it is necessary to use `unalias` before another abbreviation or macro for the same identifier can be defined. Cf. example 4.
- If a macro `f(y1,y2,...,yn)` with n arguments has been defined, it is not possible to enter a call to `f` with n arguments in its literal meaning any longer. However, entering a call to `f` with a different number of arguments is still possible. Cf. example 5.

  It is not necessary to use `unalias` before redefining an abbreviation or a macro with a different number of arguments for the identifier `f`. Any subsequent alias definition for this identifier, whether it is an abbreviation or a macro, overwrites the previous definition. See Example 4.
- An alias definition affects all kinds of input: interactive input on the command line, input via the function `input`, input from a file using `finput`, `fread`, or `read` (for the latter two only if option `Plain` is not set), and input from a string using `text2expr`. Cf. example 8.
- An alias definition has no effect on the identifier used as an alias. In particular, that identifier retains its value and its properties. The alias and the aliased object are still distinguished by the evaluator. Cf. example 6.
- Assigning a value to one of the identifiers on the left hand side of an alias definition, or deleting its value has no effect on the alias substitution, neither in the input nor in the output. See Example 7.

Example 1. We define `d` as a shortcut for `diff`:

```
>> delete f, g, x, y: alias(d = diff):
   d(sin(x), x) = diff(sin(x), x);
   d(f(x, y), x) = diff(f(x, y), x)

                        cos(x) = cos(x)

                 d(f(x, y), x) = d(f(x, y), x)
```

We define a macro `Dx(f)` for `diff(f(x), x)`. Note that `hold` does not prevent alias substitution:

```
>> alias(Dx(f) = diff(f(x), x)):
   Dx(sin); Dx(f + g); hold(Dx(f + g))
```

```
                         cos(x)

                d(f(x), x) + d(g(x), x)

                    d((f + g)(x), x)
```

After the call `unalias(d, Dx)`, no alias substitutions happen any longer:

```
>> unalias(d, Dx):
   d(sin(x), x), diff(sin(x), x), d(f(x, y), x), diff(f(x, y), x);
   Dx(sin), Dx(f + g)

      d(sin(x), x), cos(x), d(f(x, y), x), diff(f(x, y), x)

                      Dx(sin), Dx(f + g)
```

Example 2. Suppose we want to avoid typing `longhardtotypeident` and therefore define an abbreviation `a` for it:

```
>> longhardtotypeident := 10; alias(a = longhardtotypeident):

                              10
```

Since `alias` does not evaluate its arguments, `a` is now an abbreviation for `longhardtotypeident` and not for the number `10`:

```
>> type(a), type(hold(a))

                      DOM_INT, DOM_IDENT

>> a + 1, hold(a) + 1, eval(hold(a) + 1)

                         11, a + 1, 11

>> longhardtotypeident := 2:
   a + 1, hold(a) + 1, eval(hold(a) + 1)

                          3, a + 1, 3
```

However, by default alias back-substitution in the output happens for both the identifier and its current value:

```
>> 2, 10, longhardtotypeident, hold(longhardtotypeident)

                          a, 10, a, a
```

The command `Pref::alias(FALSE)` switches alias resubstitution off:

```
>> p := Pref::alias(FALSE):
   a, hold(a), 2, longhardtotypeident, hold(longhardtotypeident);
   Pref::alias(p): unalias(a):

        2, longhardtotypeident, 2, 2, longhardtotypeident
```

Example 3. Aliases are substituted and not just replaced textually. In the following example, `3*succ(u)` is replaced by `3*(u+1)`, and not by `3*u+1`, which a search-and-replace function in a text editor would produce:

```
>> alias(succ(x) = x + 1): 3*succ(u);
   unalias(succ):

                              3 u + 3
```

Example 4. We define `a` to be an abbreviation for `b`. Then the next alias definition is really an alias definition for `b`:

```
>> delete a, b:
   alias(a = b): alias(a = 2): type(a), type(b); unalias(b):

                        DOM_IDENT, DOM_INT
```

Use `unalias` first before defining another alias for the identifier `a`:

```
>> unalias(a): alias(a = 2): type(a), type(b); unalias(a):

                        DOM_INT, DOM_IDENT
```

A macro definition, however, can be overwritten immediately if the newly defined macro has a different number of arguments:

```
>> alias(a(x)=sin(x^2)): a(y); alias(a(x)=cos(x^2)):

                                 2
                              sin(y )
 Error: Illegal operand [_power];
 during evaluation of 'alias'

>> alias(a(x, y) = sin(x + y)): a(u, v); unalias(a)

                            sin(u + v)
```

Example 5. A macro definition has no effect when called with the wrong number of arguments, and the sequence of arguments is not flattened:

```
>> alias(plus(x, y) = x + y):
   plus(1), plus(3, 2), plus((3, 2));
   unalias(plus):

                      plus(1), 5, plus(3, 2)
```

Expression sequences may appear on the right hand side of an alias definition, but they have to be enclosed in parenthesis:

```
>> alias(x = (1, 2)): f := 0, 1, 2, x;
   nops(f); unalias(x):
```

```
                    0, 1, 2, 1, 2

                          5
```

Example 6. An identifier used as an abbreviation may still exist in its literal meaning inside expressions that were entered before the alias definition:

```
>> delete x: f := [x, 1]: alias(x = 1): f;
   map(f, type); unalias(x):
```

```
                        [x, x]

                 [DOM_IDENT, DOM_INT]
```

Example 7. It does not matter whether the identifier used as an alias has a value:

```
>> a := 5: alias(a = 7): 7, 5; print(a); unalias(a):
```

```
                         a, 5

                          7
```

Example 8. Alias definitions also apply to input from files or strings:

```
>> alias(a = 3): type(text2expr("a")); unalias(a)
```

```
                        DOM_INT
```

Example 9. An alias is valid for all input that is *parsed* after executing `alias`. A statement in a command line is not parsed before the previous commands in that command line have been executed. In the following example, the alias is already in effect for the second statement:

```
>> alias(a = 3): type(a); unalias(a)
```

```
DOM_INT
```

This can be changed by entering additional parentheses:

```
>> (alias(a = 3): type(a)); unalias(a)

                              DOM_INT
```

Example 10. We define `b` to be an alias for `c`, which in turn is defined to be an alias for `2`. It is recommended to avoid such chains of alias definitions beacuse of some probably unwanted effects.

```
>> alias(b=c): alias(c=2):
```

Now each `b` in the input is replaced by `c`, but no additional substitution step is taken to replace this again by `2`:

```
>> print(b)

                              c
```

On the other hand, the number `2` is replaced by `c` in every output, but that `c` is not replaced by `b`:

```
>> 2

                              c

>> unalias(c): unalias(b):
```

Example 11. When called without arguments, `alias` just displays all aliasses that are currently in effect:

```
>> alias(a = 5, F(x) = sin(x^2)):
   alias(); unalias(F, a):

 a = 5,
 F = [sin(x^2), [x]]
```

Background:

- The aliases are stored in the parser configuration table displayed by `_parser_config()`. Note that by default, alias back-substitution happens for the right hand sides of the equations in this table, but not for the indices. Use `print(_parser_config())` to display this table without alias back-substitution.
- Aliases are not in effect while a file is read using `read` or `fread` with option `Plain`. This is true in particular for all library files read with `loadproc`. Conversely, if an alias is defined in a file which is read with option `Plain`, the alias is only in effect until the file has been read completely.

`anames` – identifiers that have values or properties

`anames(`*All*`)` returns all identifiers that have values.

`anames(`*Properties*`)` returns all identifiers that have properties.

`anames(`*Protected*`)` returns all identifiers that are protected.

`anames(d)` returns all identifiers that have values from the given domain `d`.

Call(s):

- `anames(`*All* <, *User*>`)`
- `anames(`*Properties* <, *User*>`)`
- `anames(`*Protected* <, *User*>`)`
- `anames(d` <, *User*>`)`

Parameters:

`d` — a domain

Options:

All	—	get all identifiers that have values
Properties	—	get all identifiers that have properties
Protected	—	get all identifiers that are protected
User	—	exclude all system variables

Return Value: a `set` of identifiers.

Related Functions: `:=`, `_assign`, `assume`, `DOM_IDENT`

Details:

- The result returned by `anames` is a set of *unevaluated* identifiers. See example 1.
- `anames` does not take into account slots of function environments or domains. Moreover, functions of a MuPAD library are considered only if they are exported.

Option <User>:

- If the option *User* is given, only those identifiers are returned that have been assigned a value or a property, respectively, by the user.

Example 1. `anames(DOM_IDENT)` returns all identifiers which have again identifiers as values:

```
>> anames(DOM_IDENT)

  {`*`, `+`, `-`, `/`, `<`, `=`, `**`, `^`, `<=`, `<>`, `==>`}
```

The elements of the returned set are unevaluated. You can use `eval` to evaluate them:

```
>> map(%, x -> x = eval(x))

 {`*` = _mult, `+` = _plus, `<` = _less, `=` = _equal,

    `**` = _power, `^` = _power, `-` = _negate, `/` = _divide,

    `<=` = _leequal, `<>` = _unequal, `==>` = _implies}
```

Example 2. `anames(All, User)` returns all user-defined identifiers:

```
>> a := b: b := 2: c := {2, 3}:
   anames(All, User)

                          {a, b, c}
```

If the first argument is a domain, only identifiers with *values* from that domain are returned. These may differ from the identifiers whose *evaluation* belongs to the domain:

```
>> a, b;
   anames(DOM_IDENT, User);
   anames(DOM_INT, User)
```

```
                    2, 2

                    {a}

                    {b}
```

Example 3. `anames(Properties)` returns all identifiers that have been attached properties via `assume`:

```
>> assume(x > y): anames(Properties)

                    {x, y}
```

Example 4. `anames(Protected)` returns all identifiers that are protected via `protect`; since all system functions are protected, we use `anames(Protected, User)`:

```
>> protect(a): anames(Protected, User)

                    {a}
```

Changes:

- The new option *`Protected`* was introduced.

`and`, `or`, `not`, `xor`, `==>`, `<=>` – Boolean operators

`b1 and b2` represents the logical 'and' of the Boolean expressions `b1`, `b2`.

`b1 or b2` represents the non-exclusive logical 'or' of the Boolean expressions `b1`, `b2`.

`not b` represents the logical negation of the Boolean expression `b`.

`b1 xor b2` represents the exclusive logical 'or' of the Boolean expressions `b1`, `b2`.

`b1 ==> b2` represents the logical implication of the Boolean expressions `b1`, `b2`.

`b1 <=> b2` represents the logical equivalence of the Boolean expressions `b1`, `b2`.

Call(s):

- b1 <u>and</u> b2
- _and(b1, b2, ...)
- b1 <u>or</u> b2
- _or(b1, b2, ...)
- <u>not</u> b
- _not(b)
- b1 <u>xor</u> b2
- _xor(b1, b2, ...)
- b1 ==> b2
- _implies(b1, b2)
- b1 <=> b2
- _equiv(b1, b2)

Parameters:

`b, b1, b2, ...` — Boolean expressions

Return Value: a Boolean expression.

Overloadable by: `b, b1, b2, ...`

Related Functions: `_lazy_and`, `_lazy_or`, `bool`, `is`, `FALSE`, `TRUE`, `UNKNOWN`

Details:

- MuPAD uses a three state logic with the Boolean constants `TRUE`, `FALSE`, and `UNKNOWN`. These are processed as follows:

and	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

or	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

`not TRUE = FALSE`, `not FALSE = TRUE`, `not UNKNOWN = UNKNOWN` .

- The operators `xor`, `==>`, and `<=>` are defined as follows: `a xor b` is equivalent to `(a or b) and not (a and b)`; `a ==> b` is equivalent to `not a or b`; and `a <=> b` is equivalent to `(a => b) and (b => a)`.

- Boolean expressions may be composed of these constants as well as of arbitrary arithmetical expressions. Typically, equations such as `x = y` and inequalities such as `x <> y`, `x < y`, `x <= y` etc. are used to construct Boolean expressions.

- `_and(b1, b2, ...)` is equivalent to `b1 and b2 and ...`. This expression represents `TRUE` if each single expression evaluates to `TRUE`. It represents `FALSE` if at least one expression evaluates to `FALSE`. It represents `UNKNOWN` if at least one expression evaluates to `UNKNOWN` and all others evaluate to `TRUE`.

  `_and()` returns `TRUE`.

- `_or(b1, b2, ...)` is equivalent to `b1 or b2 or ...`. This expression represents `FALSE` if each single expression evaluates to `FALSE`. It represents `TRUE` if at least one expression evaluates to `TRUE`. It represents `UNKNOWN` if at least one expression evaluates to `UNKNOWN` and all others evaluate to `FALSE`.

  `_or()` returns `FALSE`.

- `_not(b)` is equivalent to `not b`.

- `_xor(b1, b2, ...)` is equivalent to `b1 xor b2 xor ...`. This expression represents `TRUE` if an odd number of operands evaluate to `TRUE` and the others evaluate to `FALSE`. It represents `FALSE` if an even number of operands evaluate to `TRUE` and the others evaluate to `FALSE`. It represents `UNKNOWN` if at least one operand evaluates to `UNKNOWN`.

- `_implies(a, b)` is equivalent to `a ==> b`.

- `_equiv(a, b)` is equivalent to `a <=> b`.

- Combinations of the constants `TRUE`, `FALSE`, `UNKNOWN` inside a Boolean expression are simplified automatically. However, symbolic Boolean subexpressions, equalities, and inequalities are not evaluated and simplified by logical operators. Use `bool` to evaluate such expressions to one of the Boolean constants. Note, however, that `bool` can evaluate inequalities `x < y`, `x <= y` etc. only if they are composed of numbers of type `Type::Real`. Cf. example 2.

  Use `simplify` with the option `logic` to simplify expressions involving symbolic Boolean subexpressions. Cf. example 3.

- The precedences of the logical operators are as follows: The operator `not` is stronger binding than `and`, i.e,

  `not b1 and b2` = `(not b1) and b2`.

  The operator `and` is stronger binding than `xor`, i.e.,

  `b1 and b2 or b3` = `(b1 and b2) xor b3`.

  The operator `xor` is stronger binding than `or`, i.e.,

  `b1 xor b2 or b3` = `(b1 xor b2) or b3`.

The operator `or` is stronger binding than `==>`, i.e.,

`b1 or b2 ==> b3` = `(b1 or b2) ==> b3`.

The operator `==>` is stronger binding than `<=>`, i.e.,

`b1 ==> b2 <=> b3` = `(b1 ==> b2) <=> b3`.

If in doubt, use brackets to make sure that the expression is parsed as desired.

- In the conditional context of `if`, `repeat`, and `while` statements, Boolean expressions are evaluated via "lazy evaluation" (see `_lazy_and`, `_lazy_or`). In any other context, all operands are evaluated.
- `_and` is a function of the system kernel.
- `_or` is a function of the system kernel.
- `_not` is a function of the system kernel.

Example 1. Combinations of the Boolean constants `TRUE`, `FALSE`, and `UNKNOWN` are simplified automatically to one of these constants:

```
>> TRUE and not (FALSE or TRUE)
```

```
FALSE
```

```
>> FALSE and UNKNOWN, TRUE and UNKNOWN
```

```
FALSE, UNKNOWN
```

```
>> FALSE or UNKNOWN, TRUE or UNKNOWN
```

```
UNKNOWN, TRUE
```

```
>> not UNKNOWN
```

```
UNKNOWN
```

Example 2. Logical operators simplify subexpressions that evaluate to the constants `TRUE`, `FALSE`, `UNKNOWN`.

```
>> b1 or b2 and TRUE
```

```
b1 or b2
```

```
>> FALSE or ((not b1) and TRUE)
```

```
                    not b1
>> b1 and (b2 or FALSE) and UNKNOWN
             UNKNOWN and b1 and b2
>> FALSE or (b1 and UNKNOWN) or x < 1
            UNKNOWN and b1 or x < 1
>> TRUE and ((b1 and FALSE) or (b1 and TRUE))
                      b1
```

However, equalities and inequalities are not evaluated:

```
>> (x = x) and (1 < 2) and (2 < 3) and (3 < 4)
        x = x and 1 < 2 and 2 < 3 and 3 < 4
```

Boolean evaluation is enforced via `bool`:

```
>> bool(%)
                     TRUE
```

Note that `bool` can compare only real numbers of syntactical type `Type::Real`:

```
>> bool(1 < 2 and PI < sqrt(10))
 Error: Can't evaluate to boolean [_less]
```

Example 3. Expressions involving symbolic Boolean subexpressions are not simplified by `and`, `or`, `not`. Simplification has to be requested explicitly via the function `simplify`:

```
>> (b1 and b2) or (b1 and (not b2)) and (1 < 2)
       b1 and b2 or b1 and not b2 and 1 < 2
>> simplify(%, logic)
                      b1
```

Example 4. The Boolean functions `_and` and `_or` accept arbitrary sequences of Boolean expressions. The following call uses `isprime` to check whether *all* elements of the given set are prime:

```
>> Set := {1987, 1993, 1997, 1999, 2001}:
   _and(isprime(i) $ i in Set)
```

```
FALSE
```

The following call checks whether *at least one* of the numbers is prime:

```
>> _or(isprime(i) $ i in Set)
```

```
TRUE
```

```
>> delete Set:
```

Changes:

- New operators `xor`, `==>`, and `<=>` have been added.

`append` – add elements to a list

`append(l, object)` adds `object` to the list `l`.

Call(s):

- `append(l, object1, object2, ...)`

Parameters:

`l`	—	a list
`object1, object2, ...`	—	arbitrary MuPAD objects

Return Value: the extended list.

Overloadable by: `l`

Related Functions: `_concat`, `_index`, `DOM_LIST`, `op`

Details:

- `append(l, object1, object2, ...)` appends `object1`, `object2`, etc. to the `list` `l` and returns the new list as the result.
- The call `append(l)` is legal and returns `l`.
- `append(l, object1, object2, ...)` is equivalent to both `[op(l), object1, object2, ...]` and `l.[object1, object2, ...]`. However, `append` is more efficient.
- The function `append` always returns a new object. The first argument remains unchanged. See example 2.
- `append` is a function of the system kernel.

Example 1. The function `append` adds new elements to the end of a list:

```
>> append([a, b], c, d)

                    [a, b, c, d]
```

If no new elements are given, the first argument is returned unmodified:

```
>> l := [a, b]: append(l)

                    [a, b]
```

The first argument may be an empty list:

```
>> append([ ], c)

                    [c]
```

Example 2. The function `append` always returns a new object. The first argument remains unchanged:

```
>> l := [a, b]: append(l, c, d), l

                    [a, b, c, d], [a, b]
```

Example 3. Users can overload `append` for their own domains. For illustration, we create a new domain `T` and supply it with an `"append"` slot, which simply adds the remaining arguments to the internal operands of its first argument:

```
>> T := newDomain("T"):
   T::append := x -> new(T, extop(x), args(2..args(0))):
```

If we now call `append` with an object of domain type `T`, the slot routine `T::append` is invoked:

```
>> e := new(T, 1, 2): append(e, 3)

                      new(T, 1, 2, 3)
```

`arcsin, arccos, arctan, arccsc, arcsec, arccot` – the inverse trigonometric functions

`arcsin(x)` represents the inverse of the sine function.

`arccos(x)` represents the inverse of the cosine function.

`arctan(x)` represents the inverse of the tangent function.

`arccsc(x)` represents the inverse of the cosecant function.

`arcsec(x)` represents the inverse of the secant function.

`arccot(x)` represents the inverse of the cotangent function.

Call(s):

- `arcsin(x)`
- `arccos(x)`
- `arctan(x)`
- `arccsc(x)`
- `arcsec(x)`
- `arccot(x)`

Parameters:

`x` — an arithmetical expression or a floating point interval

Return Value: an arithmetical expression or a floating point interval.

Overloadable by: `x`

Side Effects: When called with a floating point argument, the functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Related Functions: `sin`, `cos`, `tan`, `csc`, `sec`, `cot`

Details:

- The angle returned by these functions is measured in radians, not in degrees. E.g., the result π represents an angle of 180^o.
- All inverse trigonometric functions are defined for complex arguments.
- Floating point values are returned for floating point arguments. Floating point intervals are returned for interval arguments. Unevaluated function calls are returned for most exact arguments.
- The trigonometric functions return explicit values for arguments that are certain rational multiples of π. For these values, the inverse functions return an appropriate rational multiple of π on the main branch defined below. Cf. example 2.
- The result is expressed in terms of hyperbolic functions, if the argument is a rational multiple of `I`. Cf. example 3.
- The inverse trigonometric functions are multi-valued. The MuPAD functions return values on the main branch defined as follows. For any finite complex x:

  $y :=$ `arcsin`(x) satisfies $-\pi/2 \le \Re(y) \le \pi/2$,

  $y :=$ `arccos`(x) satisfies $0 \le \Re(y) \le \pi$,

  $y :=$ `arctan`(x) satisfies $-\pi/2 < \Re(y) < \pi/2$,

  $y :=$ `arccot`(x) satisfies $-\pi/2 < \Re(y) \le \pi/2$.

- For `arcsin` and `arccos`, the branch cuts are the real intervals $(-\infty, -1)$ and $(1, \infty)$.

  For `arctan`, the branch cuts are the intervals $(-\infty \cdot i, -i]$ and $[i, \infty \cdot i)$ on the imaginary axis.

  For `arccsc` and `arcsec`, the branch cut is the real interval $(-1, 1)$.

  For `arccot`, the branch cut is the interval $[-i, i]$ on the imaginary axis.

  The values jump when the arguments cross a branch cut. Cf. example 4.

- The functions `arccsc` and `arcsec` immediately rewrite themselves in terms of `arcsin` and `arccos`, returning `arccsc(x)=arcsin(1/x)` and `arcsec(x)=arccos(1/x)`, respectively.

Note that MuPAD's `arccot` is defined by `arccot(x)=arctan(1/x)`, although `arccot` may return an unevaluated function call and does not rewrite itself in terms of `arctan`. As a consequence of this definition, the real line crosses the branch cut and `arccot` has a jump discontinuity at the origin!

NOTE

- The float attributes are kernel functions, i.e., floating point evaluation is fast.

Example 1. We demonstrate some calls with exact and symbolic input data:

```
>> arcsin(1), arccos(1/sqrt(2)), arctan(5 + I), arccsc(1/3),
   arcsec(I), arccot(1)

     PI  PI                              PI                 PI
     --, --, arctan(5 + I), arcsin(3), -- + I arcsinh(1), --
      2   4                               2                  4
```

```
>> arcsin(-x), arccos(x + 1), arctan(1/x)

                                                    / 1 \
               -arcsin(x), arccos(x + 1), arctan| - |
                                                    \ x /
```

Floating point values are computed for floating point arguments:

```
>> arcsin(0.1234), arccos(5.6 + 7.8*I), arccot(1.0/10^20)

     0.1237153458, 0.950687977 - 2.956002937 I, 1.570796327
```

On input of floating point intervals, these functions compute floating point intervals containing the image sets:

```
>> arcsin(0...1), arccos(0...1)

      0.0 ... 1.570796327, -2.168404345e-19 ... 1.570796327
```

```
>> arcsin(2...3)

     (-1.570796327 ... 3.141592654) + (RD_NINF ... RD_INF) I
```

Note that certain types of input lead to severe overestimation, sometimes returning the whole image set of the function in question:

```
>> arccsc(-2...2);
   csc(arccsc(-2...2))

     (-3.141592654 ... 3.141592654) + (RD_NINF ... RD_INF) I

          (RD_NINF ... RD_INF) + (RD_NINF ... RD_INF) I
```

Example 2. Some special values are implemented:

```
>> arcsin(1/sqrt(2)), arccos((5^(1/2) - 1)/4), arctan(3^(1/2) - 2)

                         PI   2 PI     PI
                         --, ----, - --
                          4    5      12
```

Such simplifications occur for arguments that are trigonometric images of rational multiples of π:

```
>> sin(9/10*PI), arcsin(sin(9/10*PI))

                          1/2
                         5            PI
                         ---- - 1/4, --
                           4          10
```

```
>> cos(PI/8)/sin(PI/8), arctan(cos(PI/8)/sin(PI/8))

                          1/2     1/2
                        (2    + 2)      3 PI
                        -------------, ----
                              1/2 1/2    8
                        (2 - 2   )
```

Example 3. Arguments that are rational multiples of I are rewritten in terms of hyperbolic functions:

```
>> arcsin(5*I), arccos(5/4*I), arctan(-3*I)

                      PI
        I arcsinh(5), -- - I arcsinh(5/4), -I arctanh(3)
                       2
```

For other complex arguments unevaluated function calls without simplifications are returned:

```
>> arcsin(1/2^(1/2) + I), arccos(1 -3*I)

                   /   1/2      \
                   | 2          |
             arcsin| ---- + I |, arccos(1 - 3 I)
                   \   2        /
```

Example 4. The values jump when crossing a branch cut:

```
>> arcsin(2.0 + I/10^10), arcsin(2.0 - I/10^10)

    1.570796327 + 1.316957897 I, 1.570796327 - 1.316957897 I
```

On the branch cut, the values of `arcsin` coincide with the limit "from below" for real arguments $x > 1$. The values coincide with the limit "from above" for real $x < -1$:

```
>> arcsin(1.2), arcsin(1.2 - I/10^10), arcsin(1.2 + I/10^10)

 1.570796327 - 0.6223625037 I, 1.570796327 - 0.6223625037 I,

    1.570796327 + 0.6223625037 I

>> arcsin(-1.2), arcsin(-1.2 + I/10^10), arcsin(-1.2 - I/10^10)

 - 1.570796327 + 0.6223625037 I,

    - 1.570796327 + 0.6223625037 I,

    - 1.570796327 - 0.6223625037 I
```

Example 5. The inverse trigonometric functions can be rewritten in terms of the logarithm function with complex arguments:

```
>> rewrite(arcsin(x), ln), rewrite(arctan(x), ln)

                    2 1/2
 - I ln(I x + (1 - x )   ), 1/2 I ln(1 - I x) -

    1/2 I ln(I x + 1)
```

Example 6. Various system functions such as `diff`, `float`, `limit`, or `series` handle expressions involving the inverse trigonometric functions:

```
>> diff(arcsin(x^2), x), float(arccos(3)*arctan(5 + I))

              2 x
          -----------, - 0.06540673615 + 2.433548516 I
                4 1/2
          (1 - x )

>> limit(arcsin(x^2)/arctan(x^2), x = 0)
```

```
                              1

>> series(arctan(sin(x)) - arcsin(tan(x)), x = 0, 10)

                      7       9           11
              3   83 x    4 x    22831 x          13
          - x  - ----- - ---- - --------- + O(x  )
                   120     189     28800

>> series(arccos(2 + x), x, 3)

                                         1/2
 - arccos(2) signIm(x + 2) - 1/3 I x 3      signIm(x + 2) +

            2  1/2                     3
    1/9 I x   3     signIm(x + 2) + O(x )
```

Changes:

- floating point intervals are handled

`arcsinh`, `arccosh`, `arctanh`, `arccsch`, `arcsech`, `arccoth` – the inverse hyperbolic functions

`arcsinh(x)` represents the inverse of the hyperbolic sine function.

`arccosh(x)` represents the inverse of the hyperbolic cosine function.

`arctanh(x)` represents the inverse of the hyperbolic tangent function.

`arccsch(x)` represents the inverse of the hyperbolic cosecant function.

`arcsech(x)` represents the inverse of the hyperbolic secant function.

`arccoth(x)` represents the inverse of the hyperbolic cotangent function.

Call(s):

- `arcsinh(x)`
- `arccosh(x)`
- `arctanh(x)`
- `arccsch(x)`
- `arcsech(x)`
- `arccoth(x)`

Parameters:

`x` — an arithmetical expression or a floating point interval

Return Value: an arithmetical expression or a floating point interval

Overloadable by: `x`

Side Effects: When called with a floating point argument, the functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Related Functions: `sinh`, `cosh`, `tanh`, `csch`, `sech`, `coth`

Details:

- Theses functions are defined for complex arguments.
- Floating point values are returned for floating point arguments. Floating point intervals are returned for floating point interval arguments. Unevaluated function calls are returned for most exact arguments.
- The following special values are implemented:

  $\texttt{arcsinh}(0) = 0$, $\texttt{arccosh}(0) = i\,\pi/2$, $\texttt{arccosh}(1) = 0$,

  $\texttt{arctanh}(0) = 0$, $\texttt{arccoth}(0) = i\,\pi/2$.
- The inverse hyperbolic functions are multi-valued. The MuPAD implementations return values on the main branch defined as follows: for any finite complex x,

  $y := \texttt{arcsinh}(x)$ satisfies $-\pi/2 \le \Im(y) \le \pi/2$,

  $y := \texttt{arccosh}(x)$ satisfies $-\pi < \Im(y) \le \pi$,

  $y := \texttt{arctanh}(x)$ satisfies $-\pi/2 < \Im(y) < \pi/2$,

  $y := \texttt{arccoth}(x)$ satisfies $-\pi/2 < \Im(y) \le \pi/2$.
- The inverse hyperbolic functions are implemented according to the following relations to the logarithm function:

  `arcsinh(x) = ln(x + sqrt(x^2 + 1))`,

  `arccosh(x) = ln(x + sqrt(x^2 - 1))`,

  `arctanh(x) = (ln(1 + x) - ln(1 - x))/2`,

  `arccsch(x) = arcsinh(1/x)`,

  `arcsech(x) = arccosh(1/x)`,

  `arccoth(x) = arctanh(1/x)`.

  Cf. example 2.

- Consequently, these functions have the following branch cuts:

  For `arcsinh`, the branch cuts are the intervals $(-i \cdot \infty, -i)$ and $(i, i \cdot \infty)$ on the imaginary axis.

  For `arccosh`, the branch cuts are the real interval $(-\infty, 1)$ and the imaginary axis.

  For `arctanh`, the branch cuts are the real intervals $(-\infty, -1]$ and $[1, \infty)$.

  For `arccsch`, the branch cut is the interval $(-i, i)$ on the imaginary axis.

  For `arcsech`, the branch cuts are the real intervals $(-\infty, 0)$ and $(1, \infty)$ together with the imaginary axis.

  For `arccoth`, the branch cut is the real interval $[-1, 1]$.

  The values jump when the argument crosses a branch cut. Cf. example 3.

- The functions `arccsch` and `arcsech` immediately rewrite themselves, returning `arccsch(x)` = `arcsinh(1/x)` and `arcsech(x)` = `arccosh(1/x)`, respectively. MuPAD's `arccoth` is defined by `arccoth(x)` = `arctanh(1/x)`. However, it does not rewrite itself automatically in terms of `arctanh`.

- The float attributes are kernel functions, i.e., floating point evaluation is fast.

Example 1. We demonstrate some calls with exact and symbolic input data:

```
>> arcsinh(1), arccosh(1/sqrt(2)), arctanh(5 + I), arccsch(1/3),
   arcsech(I), arccoth(2)

                    /  1/2 \
                    | 2    |
 arcsinh(1), arccosh| ---- |, arctanh(5 + I), arcsinh(3),
                    \   2  /

    arccosh(-I), arccoth(2)

>> arcsinh(-x), arccosh(x + 1), arctanh(1/x)

                                                 / 1 \
            -arcsinh(x), arccosh(x + 1), arctanh| - |
                                                 \ x /
```

Floating point values are computed for floating point arguments:

```
>> arcsinh(0.1234), arccosh(5.6 + 7.8*I), arccoth(1.0/10^20)

    0.1230889466, 2.956002937 + 0.950687977 I, -1.570796327 I
```

Floating point intervals are returned for arguments of this type:

```
>> arccoth(0.5 ... 1.5), arcsinh(0.1234...0.12345)
```

```
 (0.2554128118 ... RD_INF) + (-1.570796327 ... 1.570796327) I,

    0.1230889466 ... 0.1231385701
```

The inverse of the hyperbolic tangent function has real values only in the interval $(-1, 1)$:

```
>> arctanh(-1/2...0), arctanh(2...3)

 -0.5493061444 ... 9.215718467e-19,

    (0.2027325540 ... 0.6931471806) +

    (-1.570796327 ... 1.570796327) I
```

Example 2. The inverse hyperbolic functions can be rewritten in terms of the logarithm function:

```
>> rewrite(arcsinh(x), ln), rewrite(arctanh(x), ln)

                   2     1/2   ln(x + 1)   ln(1 - x)
           ln(x + (x  + 1)   ), --------- - ---------
                                    2           2
```

Example 3. The values jump when crossing a branch cut:

```
>> arctanh(2.0 + I/10^10), arctanh(2.0 - I/10^10)

   0.5493061443 + 1.570796327 I, 0.5493061443 - 1.570796327 I
```

On the branch cut, the values of `arctanh` coincide with the limit "from below" for real arguments $x > 1$. The values coincide with the limit "from above" for real $x < -1$:

```
>> arctanh(1.2), arctanh(1.2 - I/10^10), arctanh(1.2 + I/10^10)

 1.198947636 - 1.570796327 I, 1.198947636 - 1.570796327 I,

    1.198947636 + 1.570796327 I

>> arctanh(-1.2), arctanh(-1.2 + I/10^10), arctanh(-1.2 - I/10^10)

 - 1.198947636 + 1.570796327 I, - 1.198947636 + 1.570796327 I,

    - 1.198947636 - 1.570796327 I
```

Example 4. Various system functions such as `diff`, `float`, `limit`, or `series` handle expressions involving the inverse hyperbolic functions:

```
>> diff(arcsinh(x^2), x), float(arccosh(3)*arctanh(5 + I))

                  2 x
            -----------, 0.3427241326 + 2.698556745 I
              4     1/2
            (x  + 1)

>> limit(arcsinh(x)/arctanh(x), x = 0)

                                1

>> series(arctanh(sinh(x)) - arcsinh(tanh(x)), x = 0, 10)

                        7       9           11
              3     83 x     4 x     22831 x         13
             x  + ----- - ---- + --------- + O(x   )
                     120     189      28800
```

Changes:

- Floating point intervals are handled.

`arg` – the argument (polar angle) of a complex number

`arg(z)` returns the argument of the complex number `z`.

`arg(x, y)` returns the argument of the complex number with real part `x` and imaginary part `y`.

Call(s):

- `arg(z)`
- `arg(x, y)`

Parameters:

`z`	—	arithmetical expression
`x, y`	—	arithmetical expressions representing real numbers

Return Value: an arithmetical expression.

Overloadable by: `x, z`

Side Effects: When called with floating point arguments, the function is sensitive to the environment variable `DIGITS` which determines the numerical working precision. Properties of identifiers are taken into account.

Related Functions: `arctan`, `Im`, `Re`, `rectform`

Details:

- The argument of a non-zero complex number $z = x + i\,y = |z|\,e^{i\,\phi}$ is its real polar angle ϕ. `arg(x,y)` represents the principal value $\phi \in (-\pi, \pi]$. For $x \neq 0$, $y \neq 0$, it is given by

$$\texttt{arg}(x, y) = \texttt{arctan}\left(\frac{y}{x}\right) + \frac{\pi}{2}\;\texttt{sign}(y)\;(1 - \texttt{sign}(x)).$$

- An error occurs if `arg` is called with two arguments and either one of the arguments `x`, `y` is a non-real numerical value. Symbolic arguments are assumed to be real.
- On the other hand, if `arg` is called with only one argument `x + I*y`, it is not assumed that `x` and `y` are real.
- A floating point number is returned if one argument is given which is a floating point number; or if two arguments are given, both of them are numerical and at least one of them is a floating point number.
- If the sign of the arguments can be determined, then the result is expressed in terms of `arctan`. Cf. example 2. Otherwise, a symbolic call of `arg` is returned. Numerical factors are eliminated from the first argument. Cf. example 3.
- A symbolic call to `arg` returned has only one argument.
- The call `arg(0,0)`, or equivalently `arg(0)`, returns `0`.
- An alternative representation is $\texttt{arg}(x, y) = -i\ln(z/|z|) = -i\ln(\texttt{sign}(z))$. Cf. example 4.

Example 1. We demonstrate some calls with exact and symbolic input data:

```
>> arg(2, 3), arg(x, 4), arg(4, y), arg(x, y), arg(10, y + PI)

                                     / y \
 arctan(3/2), arg(x + 4 I), arctan| - |, arg(x + I y),
                                     \ 4 /

         /  y   PI \
   arctan| -- + -- |
         \ 10   10 /
```

If arg is called with two arguments, the arguments are implicitly assumed to be real, which allows some additional simplifications compared to a call with only one argument:

```
>> arg(1, y), arg(1 + I*y)

                    arctan(y), arg(I y + 1)

>> arg(x, infinity), arg(-infinity, 3), arg(-infinity, -3)

                          PI
                          --, PI, -PI
                           2
```

Floating point values are computed for floating point arguments:

```
>> arg(2.0, 3), arg(2, 3.0), arg(10.0^100, 10.0^(-100))

              0.9827937232, 0.9827937232, 1.0e-200
```

Example 2. arg reacts to properties of identifiers set via assume:

```
>> assume(x > 0): assume(y < 0): arg(x, y)

                                / y \
                          arctan| - |
                                \ x /

>> assume(x < 0): assume(y > 0): arg(x, y)

                                  / y \
                       PI + arctan| - |
                                  \ x /

>> assume(x <> 0): arg(x, 3)

                 PI (1 - sign(x))         / 3 \
                 ---------------- + arctan| - |
                         2                \ x /

>> unassume(x), unassume(y):
```

Example 3. Certain simplifications may occur in unevaluated calls. In particular, numerical factors are eliminated from the first argument:

```
>> arg(3*x, 9*y), arg(-12*sqrt(2)*x, 12*y)

                                               1/2
                arg(x + 3 I y), arg(I y - x 2    )
```

Example 4. Use `rewrite` to convert symbolic calls of `arg` to the logarithmic representation:

```
>> rewrite(arg(x, y), ln)

                          /     x + I y    \
                    - I ln| ------------ |
                          \ abs(x + I y) /
```

Example 5. System functions such as `float`, `limit`, or `series` handle expressions involving `arg`:

```
>> limit(arg(x, x^2/(1+x)), x = infinity)

                              PI
                              --
                               4
```

```
>> series(arg(x, x^2), x = 1, 4, Real)

                                  2           3
         PI    (x - 1)    (x - 1)     (x - 1)              4
         -- + ------- - -------- + -------- + O((x - 1) )
          4      2          4         12
```

Changes:

- `arg` may now also be called with only one argument.

---

args – access procedure parameters

`args(0)` returns the number of parameters of the current procedure.

`args(i)` returns the value of the `i`th parameter of the current procedure.

Call(s):

- `args()`
- `args(0)`
- `args(i)`
- `args(i..j)`

Parameters:

`i, j` — positive integers

Return Value: `args(0)` returns a nonnegative integer. All other calls return an arbitrary MuPAD object or a `sequence` of such objects.

Related Functions: `context`, `DOM_PROC`, `DOM_VAR`, `Pref::typeCheck`, `proc`, `procname`, `testargs`

Details:

- `args` accesses the actual parameters of a procedure and can only be used in procedures. It is mainly intended for procedures with a variable number of arguments, since otherwise parameters can simply be accessed by their names.
- `args()` returns an expression sequence of all actual parameters.
- `args(0)` returns the number of actual parameters.
- `args(i)` returns the value of the `i`th parameter.
- `args(i..j)` returns an expression sequence containing the `i`th up to the `j`th parameter.
- In procedures with option `hold`, `args` returns the parameters without further `evaluation`. Use `context` or `eval` to enforce a subsequent evaluation. See example 2.
- `procname(args())` returns a symbolic function call of the current procedure with evaluated arguments.
- Assigning values to formal parameters of a procedure changes the result of `args`. Cf. example 4. `args(0)` remains unchanged.
- `args` is a function of the system kernel.

Example 1. This example demonstrates the various ways of using `args`:

```
>> f := proc() begin
     print(Unquoted, "number of arguments" = args(0)):
     print(Unquoted, "sequence of all arguments" = args()):
     if args(0) > 0 then
       print(Unquoted, "first argument" = args(1)):
     end_if:
     if args(0) >= 3 then
       print(Unquoted, "second, third argument" = args(2..3)):
     end_if:
   end_proc:
```

```
>> f():
```

number of arguments = 0

sequence of all arguments = null()

```
>> f(42):
```

number of arguments = 1

sequence of all arguments = 42

first argument = 42

```
>> f(a, b, c, d):
```

number of arguments = 4

sequence of all arguments = (a, b, c, d)

first argument = a

second, third argument = (b, c)

Example 2. `args` does not evaluate the returned parameters in procedures with the option `hold`. Use `context` to achieve this:

```
>> f := proc()
     option hold;
   begin
     args(1), context(args(1))
   end_proc:
```

```
>> delete x, y: x := y: y := 2: f(x)
```

x, 2

Example 3. We use `args` to access parameters of a procedure with an arbitrary number of arguments:

```
>> f := proc() begin
     args(1) * _plus(args(2..args(0)))
   end_proc:
   f(2, 3), f(2, 3, 4)
```

6, 14

Example 4. Assigning values to formal parameters affects the behavior of `args`. In the following example, `args` returns the value `4`, which is assigned inside the procedure, and not the value `1`, which is the argument of the procedure call:

```
>> f := proc(a) begin a := 4; args() end_proc:
   f(1)

                              4
```

`array` – create an array

`array(m1..n1, m2..n2, ...)` creates an array with uninitialized entries, where the first index runs from m_1 to n_1, the second index runs from m_2 to n_2, etc.

`array(m1..n1, m2..n2, ..., list)` creates an array with entries initialized from `list`.

Call(s):

- array(m1..n1 <, m2..n2, ...>)
- array(m1..n1, <m2..n2, ...,> index1 = entry1, index2 = entry2, ...)
- array(m1..n1, <m2..n2, ...,> list)

Parameters:

`m1, n1, m2, n2, ...`	—	the boundaries: integers
`index1, index2, ...`	—	a sequence of integers defining a valid array index
`entry1, entry2, ...`	—	arbitrary objects
`list`	—	a list, possibly nested

Return Value: an object of type `DOM_ARRAY`.

Related Functions: `_assign`, `_index`, `assignElements`, `delete`, `DOM_ARRAY`, `DOM_LIST`, `DOM_TABLE`, `indexval`, `matrix`, `table`

Details:

- Arrays are container objects for storing data. In contrast to `tables`, the indices must be sequences of integers. While `tables` may grow in size dynamically, the number of entries in an array is fixed.

- For an array `A`, say, and a sequence of integers `index` forming a valid array index, an indexed call `A[index]` returns the corresponding entry. If the entry is uninitialized, then the indexed expression `A[index]` is returned. See examples 1 and 4.
- An indexed assignment of the form `A[index]:=entry` initializes or overwrites the entry corresponding to `index`. See examples 1 and 4.
- `array` creates an array. The boundaries must satisfy $m_1 \leq n_1$, $m_2 \leq n_2$, etc. The dimension of the resulting array is the number of given range arguments; at least one range argument is mandatory. The total number of entries of the resulting array is $(m_1 - n_1 + 1)(m_2 - n_2 + 1) \cdots$.
- If only index range arguments are given, then an array with uninitialized entries is created. See example 1.
- If equations of the form `index=entry` are present, then the array entry corresponding to `index` is initialized with `entry`. This is useful for selectively initializing some particular array entries.

  Each index must be a valid array index of the form `i1` for one-dimensional arrays and `(i1,i2,..)` for higher-dimensional arrays, where `i1,i2,...` are integers within the valid boundaries, satisfying $m_1 \leq i_1 \leq n_1$, $m_2 \leq i_2 \leq n_2$, etc., and the number of integers in `index` matches the dimension of the array.
- If the argument `list` is present, then the resulting array is initialized with the entries from `list`. This is useful for initializing all array entries at once. The structure of the list must match the structure of the array exactly, such that the nesting depth of the list is greater or equal to the dimension of the array and the number of list entries at the kth nesting level is equal to the size of the kth index range. Cf. example 6.
- A call of the form `delete A[index]` deletes the entry corresponding to `index`, so that it becomes uninitialized again. See example 4.
- NOTE Internally, uninitialized array entries have the value `NIL`. Thus assigning `NIL` to an array entry has the same effect as deleting it via `delete`, and afterwards an indexed call of the form `A[index]` returns the symbolic expression `A[index]`, and not `NIL`, as one might expect. See example 4.
- A one-dimensional array is printed as a row vector. The index corresponds to the column number.
- A two-dimensional array is printed as a matrix. The first index corresponds to the row number and the second index corresponds to the column number.
- A one- or two-dimensional array that is so big that it would exceed the maximal output width `TEXTWIDTH` is printed in the form

`array(m1..n1, m2..n2, ..., index1 = entry1, index2 = entry2, ...)` See example 8. The same is true for arrays of dimension greater than two. See examples 5 and 6.

- Arithmetic operations are not defined for arrays. Use `matrix` to create one-dimensional vectors and two-dimensional matrices in the mathematical sense.
- If an array is evaluated, it is only returned. The evaluation does not map recursively on the array entries. This is due to performance reasons. You have to `map` the function `eval` explicitly on the array in order to fully evaluate its entries. See example 7.
- `array` is a function of the system kernel.

Example 1. We create an uninitialized one-dimensional array with indices ranging from 2 to 4:

```
>> A := array(2..4)

                         +-                 -+
                         | ?[2], ?[3], ?[4] |
                         +-                 -+
```

The question marks in the output indicate that the array entries are not initialized. We set the middle entry to 5 and last entry to `"MuPAD"`:

```
>> A[3] := 5: A[4] := "MuPAD": A

                         +-                 -+
                         | ?[2], 5, "MuPAD" |
                         +-                 -+
```

You can access array entries via indexed calls. Since the entry `A[2]` is not initialized, the symbolic expression `A[2]` is returned:

```
>> A[2], A[3], A[4]

                           A[2], 5, "MuPAD"
```

We can initialize an array already when creating it by passing initialization equations to `array`:

```
>> A := array(2..4, 3 = 5, 4 = "MuPAD")

                         +-                 -+
                         | ?[2], 5, "MuPAD" |
                         +-                 -+
```

We can initialize all entries of an array when creating it by passing a list of initial values to `array`:

```
>> array(2..4, [PI, 5, "MuPAD"])

                        +-               -+
                        | PI, 5, "MuPAD" |
                        +-               -+
```

Example 2. Array boundaries may be negative integers as well:

```
>> A := array(-1..1, [2, sin(x), FAIL])

                        +-                -+
                        | 2, sin(x), FAIL |
                        +-                -+

>> A[-1], A[0], A[1]

                          2, sin(x), FAIL
```

Example 3. The `$` operator may be used to create a sequence of initialization equations:

```
>> array(1..8, i = i^2 $ i = 1..8)

                  +-                               -+
                  | 1, 4, 9, 16, 25, 36, 49, 64 |
                  +-                               -+
```

Equivalently, you can use the `$` operator to create an initialization list:

```
>> array(1..8, [i^2 $ i = 1..8])

                  +-                               -+
                  | 1, 4, 9, 16, 25, 36, 49, 64 |
                  +-                               -+
```

Example 4. We create a 2×2 matrix as a two-dimensional array:

```
>> A := array(1..2, 1..2, (1, 2) = 42, (2, 1) = 1 + I)

                      +-                    -+
                      |  ?[1, 1],     42     |
                      |                      |
                      |   1 + I,   ?[2, 2]   |
                      +-                    -+
```

Internally, array entries are stored in a linearized form. They can be accessed in this form via `op`. Uninitialized entries internally have the value `NIL`:

```
>> op(A, 1), op(A, 2), op(A, 3), op(A, 4)

                      NIL, 42, 1 + I, NIL
```

Note the difference to the indexed access:

```
>> A[1, 1], A[1, 2], A[2, 1], A[2, 2]

                  A[1, 1], 42, 1 + I, A[2, 2]
```

We can modify an array entry by an indexed assignment:

```
>> A[1, 1] := 0: A[1, 2] := 5: A

                      +-                 -+
                      |    0,       5     |
                      |                   |
                      |  1 + I, ?[2, 2]   |
                      +-                 -+
```

You can delete the value of an array entry via `delete`. Afterwards, it is uninitialized again:

```
>> delete A[2, 1]: A[2, 1], op(A, 3)

                         A[2, 1], NIL
```

Assigning `NIL` to an array entry has the same effect as deleting it:

```
>> A[1, 2] := NIL: A[1, 2], op(A, 2)

                         A[1, 2], NIL
```

Example 5. We define a three-dimensional array with index values between 1 and 8 in each of the three dimensions and initialize two of the entries via initialization equations:

```
>> A := array(1..8, 1..8, 1..8,
              (1, 1, 1) = 111,
              (8, 8, 8) = 888)

                    array(1..8, 1..8, 1..8,
                      (1, 1, 1) = 111,
                      (8, 8, 8) = 888
                    )

>> A[1, 1, 1], A[1, 1, 2]

                        111, A[1, 1, 2]
```

Example 6. A nested list may be used to initialize a two-dimensional array. The inner lists are the rows of the created matrix:

```
>> array(1..2, 1..3, [[1, 2, 3], [4, 5, 6]])
```

```
                                   +-         -+
                                   |  1, 2, 3  |
                                   |           |
                                   |  4, 5, 6  |
                                   +-         -+
```

We create a three-dimensional array and initialize it from a nested list of depth three. The outer list has two entries for the first dimension. Each of these entries is a list with three entries for the second dimension. Finally, the innermost lists each have one entry for the third dimension:

```
>> array(2..3, 1..3, 1..1,
         [
           [ [1], [2], [3] ],
           [ [4], [5], [6] ]
         ])
```

```
                    array(2..3, 1..3, 1..1,
                      (2, 1, 1) = 1,
                      (2, 2, 1) = 2,
                      (2, 3, 1) = 3,
                      (3, 1, 1) = 4,
                      (3, 2, 1) = 5,
                      (3, 3, 1) = 6
                    )
```

Example 7. If an array is evaluated, it is only returned. The evaluation does not map recursively on the array entries. Here, the entries `a` and `b` are not evaluated:

```
>> A := array(1..2, [a, b]):
   a := 1:  b := 2:
   A, eval(A)
```

```
                              +-     -+  +-     -+
                              | a, b |,  | a, b |
                              +-     -+  +-     -+
```

Due to the special evaluation of arrays the index operator evaluates array entries after extracting them from the array:

```
>> A[1], A[2]
```

```
1, 2
```

You have to map the function eval explicitly on the array in order to fully evaluate its entries:

```
>> map(A, eval)

                        +-      -+
                        | 1, 2 |
                        +-      -+
```

Example 8. A two-dimensional array is usually printed in matrix form:

```
>> A := array(1..4, 1..4,
              (1, 1) = 11,
              (4, 4) = 44)

            +-                                    -+
            |    11,   ?[1, 2], ?[1, 3], ?[1, 4]  |
            |                                      |
            |  ?[2, 1], ?[2, 2], ?[2, 3], ?[2, 4]  |
            |                                      |
            |  ?[3, 1], ?[3, 2], ?[3, 3], ?[3, 4]  |
            |                                      |
            |  ?[4, 1], ?[4, 2], ?[4, 3],    44    |
            +-                                    -+
```

If the output does not fit into TEXTWIDTH, a more compact output is used:

```
>> TEXTWIDTH := 20:
   A;
   delete TEXTWIDTH:

  array(1..4, 1..4,
    (1, 1) = 11,
    (4, 4) = 44
  )
```

---

assert – assertions for debugging

The statement assert(cond) declares that the condition cond holds true at the moment when the statement is evaluated. By default, MuPAD does not care about assertions. After setting testargs(TRUE), however, MuPAD checks every assertion and stops with an error if boolean evaluation of cond does not give TRUE.

Call(s):

- ```assert(cond)```

Parameters:

`cond` — a boolean expression

Return Value: `assert` returns `TRUE` or raises an error.

Related Functions: `testargs`

Details:

- Assertions are a major debugging tool for programmers: by stating frequently what she thinks to have achieved, a programmer makes it easy for herself to detect the first unintended intermediate result.

Example 1. Suppose we want to write a function `f` that takes an integer as its argument and returns 0 if that integer is a multiple of 3, and 1 otherwise. One idea how to code this could be the following: given an integer n, n modulo 3 must be equal to one of -1, 1, or 0. In any case, `abs(n mod 3)` should do what we want:

```
>> f := proc(n: DOM_INT): DOM_INT
   local k: DOM_INT;
   begin
     k := n mod 3;
     assert(k = 1 or k = -1 or k = 0);
     abs(k)
   end_proc
```

```
                        proc f(n) ... end
```

Checking assertions is switched on or off using `testargs`:

```
>> oldtestargs := testargs(): testargs(FALSE): f(5)
```

```
                                2
```

The result does not equal `1`. Ror debugging purposes, we switch on assertion checking:

```
>> testargs(TRUE): f(5)
```

```
 Error: Assertion k = 1 or k = -1 or k = 0 failed [f]
```

This shows that the local variable `k` must have gotten a wrong value. Indeed, when writing our program we overlooked the difference between `mod` and the symmetric remainder given by `mods`.

```
>> testargs(oldtestargs):
```

Changes:

- `assert` is a new function.

`assign` – perform assignments given as equations

For each equation in a list, a set, or a table of equations `L`, `assign(L)` evaluates both sides of the equation and assigns the evaluated right hand side to the evaluated left hand side.

`assign(L, S)` does the same, but only for those equations whose left hand side is in the set `S`.

Call(s):

- `assign(L)`
- `assign(L, S)`

Parameters:

`L` — a list, a set, or a table of equations
`S` — a set

Return Value: `L`.

Related Functions: `:=`, `_assign`, `assignElements`, `delete`, `evalassign`

Details:

- Since the arguments of `assign` are evaluated, the *evaluation* of the left hand side of each equation in `L` must be an admissible left hand side for an assignment. See the help page of the assignment operator `:=` for details.
- Several assignments are performed from left to right. See example 4.
- `assign` can be conveniently used after a call to `solve` to assign a particular solution of a system of equations to the unknowns. See example 5.

Example 1. We assign values to the three identifiers `B1,B2,B3`:

```
>> delete B1, B2, B3:
   assign([B1 = 42, B2 = 13, B3 = 666]): B1, B2, B3

                        42, 13, 666
```

We specify a second argument to carry out only those assignments with left hand side `B1`:

```
>> delete B1, B2, B3:
   assign([B1 = 42, B2 = 13, B3 = 666], {B1}): B1, B2, B3
```

42, B2, B3

The first argument may also be a table of equations:

```
>> delete B1, B2, B3:
   assign(table(B1 = 42, B2 = 13, B3 = 666)): B1, B2, B3
```

42, 13, 666

Example 2. Unlike `_assign`, `assign` evaluates the left hand sides:

```
>> delete a, b: a := b: assign({a = 3}): a, b
```

3, 3

```
>> delete a, b: a := b: a := 3: a, b
```

3, b

Example 3. The object assigned may also be a sequence:

```
>> assign([X=(2,7)])
```

[X = (2, 7)]

```
>> X
```

2, 7

Example 4. The assignments are carried out one after another, from left to right. Since the right hand side is evaluated, the identifier `C` gets the value `3` in the following example:

```
>> assign([B=3, C=B])
```

[B = 3, C = B]

```
>> level(C,1)
```

3

Example 5. When called for an algebraic system, `solve` often returns a set of lists of assignments. `assign` can then be used to assign the solutions to the variables of the system:

```
>> sys:={x^2+y^2=2, x+y=5}:
   S:= solve(sys)

                      1/2               1/2
 {[x = 5/2 - 1/2 I 21    , y = 1/2 I 21    + 5/2],

                  1/2                          1/2
    [x = 1/2 I 21    + 5/2, y = 5/2 - 1/2 I 21   ]}
```

We want to check whether the first solution is really a solution:

```
>> assign(S[1]): sys

                            1/2 2             1/2       2
    {5 = 5, (5/2 - 1/2 I 21    )  + (1/2 I 21    + 5/2)  = 2}
```

Things become clearer if we use floating point evaluation:

```
>> float(sys)

            {5.0 = 5.0, 2.0 - 8.67361738e-19 I = 2.0}
```

`assignElements` – assign values to entries of an array, a list, or a table

`assignElements(L, [index1] = value1, [index2] = value2, ...)` returns a copy of `L` with `value1` stored at `index1`, `value2` stored at `index2`, etc.

Call(s):

- `assignElements(L, [index1] = value1, [index2] = value2, ...)`
- `assignElements(L, [[index1], value1], [[index2], value2], ...)`

Parameters:

`L`	—	an array, a list, or a table
`index1, index2, ...`	—	valid indices for `L`
`value1, value2, ...`	—	any MuPAD objects

Return Value: an object of the same type as `L`.

Related Functions: `:=`, `_assign`, `_index`, `array`, `assign`, `delete`, `DOM_ARRAY`, `DOM_LIST`, `DOM_TABLE`, `evalassign`, `table`

Details:

- `R:=assignElements(L,[index1]=value1,[index2]=value2,...)` has the same effect as the sequence of assignments `R:=L: R[index1]:=value1: R[index2]:=value2: ... R`, but is more efficient.
- `assignElements` returns a modified copy of its first argument, which remains unchanged. See example 1.
- The second variant of the `assignElements` call, with lists instead of equations, is equivalent to the first variant. In fact, both equations and lists may be mixed in a single call. See example 1.
- All assignments are performed simultaneously, i.e., the order of the arguments is irrelevant. See example 3.
- All rules for indexed assignments apply, in particular with respect to the validity of indices. If `L` is a list, the indices must be positive integers not exceeding the length of `L`. If `L` is an array, the indices must be (sequences of) integers matching the dimension and lying within the valid ranges of the array. If `L` is a table, the indices may be arbitrary objects.
- `assignElements` is a function of the system kernel.

Example 1. Assignments may given as equations or lists, and both forms may be mixed in a single call:

```
>> L := array(1..3, [3, 4, 5]);
   assignElements(L, [1] = one, [2] = two, [3] = three);
   assignElements(L, [[1], one], [[2], two], [[3], three]);
   assignElements(L, [1] = one, [[2], two], [3] = three);

                        +-       -+
                        | 3, 4, 5 |
                        +-       -+

                    +-               -+
                    | one, two, three |
                    +-               -+

                    +-               -+
                    | one, two, three |
                    +-               -+

                    +-               -+
                    | one, two, three |
                    +-               -+
```

The array L itself is not modified by assignElements:

```
>> L
```

```
+-         -+
| 3, 4, 5 |
+-         -+
```

Example 2. Sequences, too, may be assigned as values to array elements, but they must be put in parentheses:

```
>> R := assignElements(array(1..2), [1] = (1, 7), [2] = PI)
```

```
+-          -+
| 1, 7, PI |
+-          -+
```

```
>> [R[1]], [R[2]]
```

```
[1, 7], [PI]
```

Example 3. The sequence generator $ is useful to create sequences of assignments:

```
>> L := [i $ i = 1..10];
   assignElements(L, [i] = L[i] + L[i + 1] $ i = 1..9)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

[3, 5, 7, 9, 11, 13, 15, 17, 19, 10]
```

The order of the arguments is irrelevant:

```
>> assignElements(L, [10 - i] = L[10 - i] + L[11 - i] $ i = 1..9)
```

```
[3, 5, 7, 9, 11, 13, 15, 17, 19, 10]
```

Example 4. The indices of a table may be arbitrary objects, for example, strings:

```
>> assignElements(table(), [expr2text(i)] = i^2 $ i = 1..4)
```

```
table(
  "4" = 16,
  "3" = 9,
  "2" = 4,
  "1" = 1
)
```

Example 5. For arrays of dimension greater than one, the indices are sequences of as many integers as determined by the dimension of the array:

```
>> assignElements(array(1..3, 1..3),
     ([i, j] = i + j $ i = 1..3) $ j = 1..3)

                          +-         -+
                          |  2, 3, 4  |
                          |           |
                          |  3, 4, 5  |
                          |           |
                          |  4, 5, 6  |
                          +-         -+
```

---

assume – attach a property to an identifier

`assume(x, prop)` attaches the property `prop` to the identifier `x`.

`assume(prop)` sets a "global property" that is valid for all identifiers.

Call(s):

- `assume(x, prop` $<$`, _and_or`$>$`)`
- `assume(prop` $<$`, _and_or`$>$`)`
- `assume(y rel z` $<$`, _and_or`$>$`)`

Parameters:

`x`	—	an identifier or one of the expressions `Re(u)` or `Im(u)` with an identifier `u`
`prop`	—	a property
`_and_or`	—	either `_and` or `_or`. Without this optional argument, any previously attached property is overwritten by the new property. With `_and` or `_or`, existing properties are not deleted but logically combined with the new property by 'and' or 'or', respectively.
`y, z`	—	arithmetical expressions
`rel`	—	one of the relational operators `<`, `<=`, `=`, `<>`, `>=`, `>`

Return Value: a property of type `Type::Property`.

Related Functions: `_assign`, `anames`, `getprop`, `is`, `property::hasprop`, `property::implies`, `Type`, `Type::Property`, `unassume`

Details:

- Properties represent subsets of the complex plane. Attaching a property `prop` to an identifier `x` corresponds to the statement '`x` represents a number of the set `prop`'. Various predefined properties are installed in the `Type` library. Request `?properties` for a list of all available properties.

  By default, library functions regard identifiers as symbols representing complex numbers. For this reason, certain expressions such as `sign(1 + x^2)` cannot be simplified in any way unless `x` is restricted to some subset of the complex plane. E.g., if `x` is assumed to be a real number, the expression may be simplified to `sign(1 + x^2)` = `1`.

  Thus, properties help to simplify expressions. Various system functions react to properties and yield simpler results. Cf. example 7.

  Properties of identifiers are set via `assume`. Properties of expressions are queried by the functions `getprop` and `is`.

- `assume(x, prop)` attaches the property `prop` to the identifier `x`. Without `_and_or`, existing properties of `x` are overwritten.

- If the optional argument `_and` or `_or` is given, existing properties of `x`, `y`, or `z`, if any, are not overwritten, but logically combined with the new property via 'and' or 'or'. The resulting property is then attached to `x`, `y`, or `z`. Cf. example 2.

- It may happen that the resulting property cannot be represented explicitly. In this case, a weaker property is attached instead. Cf. example 8.

- `assume(Re(u), prop)` and `assume(Im(u), prop)` attach the property to the real and the imaginary part of the identifier `u`, respectively. Cf. example 3.

- In `assume(y rel z)`, at least one of `y` or `z` must be an identifier or of the form `Re(u)` or `Im(u)`. The other one may be an arbitrary arithmetical expression.

  The property representing the relation is attached to the identifier(s) `y` and/or `z`. In particular, if both `y` and `z` are identifiers or of the form `Re(u)`, `Im(u)`, then any existing property of both `y` and `z` are overwritten, unless `_and_or` is specified.

  If `rel` is one of `<`, `>`, `<=` or `>=`, and `y` or `z` is an identifier or `Re(u)`, `Im(u)`, then the property `Type::Real` is implicitly attached to `y` and/or `z`.

  Cf. example 4.

- The call `assume(prop <, _and_or>)` defines a "global property" `prop` that is assumed to hold for *all* identifiers. When querying properties of expressions (e.g., via `is`), this global property is logically combined with the properties of individual identifiers via 'and'.

The argument `_and_or` indicates that an existing global property is combined logically with the new global property.

The protected identifier `Global` is used to store global properties.

The calls `assume(prop <, _and_or>)` and `assume(Global, prop <, _and_or>)` are equivalent.

Cf. example 5.

- Properties of an identifier `x` are deleted via `unassume(x)` or `delete x`. The global property is deleted via `unassume()` or `unassume(Global)` (this does not affect the individual properties of identifiers).

  When assigning a value to an identifier with properties, the assigned value needs not be consistent in any way with previously assigned properties. Properties are overwritten by an assignment. Cf. example 6.

Example 1. The following command marks the identifier `n` as an integer:

```
>> assume(n, Type::Integer)

                         Type::Integer
```

MuPAD can now derive that `n^2` is a nonnegative integer:

```
>> getprop(n^2), is(n^2, Type::NonNegInt)

                     Type::NonNegInt, TRUE
```

Also other system functions react to this property:

```
>> abs(n^2 + 1), simplify(sin(2*n*PI))

                            2
                           n  + 1, 0

>> delete n:
```

Example 2. Using `_and` or `_or`, existing properties are not deleted, but combined with new properties:

```
>> assume(n, Type::NonNegInt)

                        Type::NonNegInt

>> assume(n, Type::NegInt, _or)

                         Type::Integer

>> assume(n, Type::Positive, _and)

                         Type::PosInt

>> delete n:
```

Example 3. Properties of the real and the imaginary part of an identifier can be defined separately:

```
>> assume(Re(z) > 0), assume(Im(z) < 0, _and)
```

```
Re(.) > 0, Re(.) > 0 and Im(.) < 0
```

```
>> abs(Re(z)), sign(Im(z))
```

```
Re(z), -1
```

```
>> is(z, Type::Real), is(z > 0)
```

```
FALSE, FALSE
```

```
>> delete z:
```

Example 4. Assuming relations such as `x > y` affects the properties of both identifiers:

```
>> assume(x > y)
```

```
< x
```

Properties can be queried by `getprop`. Both `x` and `y` have properties:

```
>> getprop(x), getprop(y)
```

```
> y, < x
```

In the next command, `_and` is used to prevent that the previous property of `y` is deleted: `y` is assumed to be greater than `0` *and* less than `x`:

```
>> assume(y > 0, _and)
```

```
]0, x[
```

```
>> is(x^2 >= y^2)
```

```
TRUE
```

The second `assume` in the next example *without* the operator `_and` would have overwritten the property '`> 0`' of `x`. With `_and`, the assumption `x >= 0` stays valid:

```
>> unassume(y):
   assume(x >= 0): assume(y >= x, _and): is(y >= 0)
```

```
TRUE
```

Relations such as `x > y` imply that the involved identifiers are real:

```
>> is(x, Type::Real), is(y, Type::Real)

                    TRUE, TRUE

>> delete x, y:
```

In the following example, one side of the given relation is not an identifier but an expression. Consequently, the property is attached only to the identifier `x`:

```
>> assume(x > 1/y)

                      > 1/y

>> getprop(x), getprop(y)

                    > 1/y, y

>> delete x:
```

Example 5. In the next example, a global property is defined:

```
>> assume(Type::NonNegative)

                Type::NonNegative
```

Now, any identifier is assumed to be nonnegative and real:

```
>> Re(x), Im(y), sign(1 + z^2)

                     x, 0, 1
```

Individual assumptions may be attached to identifiers independent of the global property:

```
>> assume(x, Type::Integer)

                  Type::Integer
```

Deductions of properties via `getprop` or `is` combine individual properties with the global property:

```
>> getprop(x), is(x < 0)

              Type::NonNegInt, FALSE
```

Also the global property can be modified using `_and` and `_or`:

```
>> assume(Type::Negative, _or)
```

```
Type::Real
```

To define a relation as a global property, the identifier `Global` must be used:

```
>> assume(Global > 0): is(x + y + z > 0)
```

```
TRUE
```

The global property can only be deleted with the call `unassume()`:

```
>> delete x: unassume():
```

Example 6. `_assign` and `:=` do not check the properties of an identifier. All properties are overwritten:

```
>> assume(a > 0): a := -2: a, getprop(a)
```

```
-2, -2
```

```
>> delete a:
```

Example 7. Some system functions take properties of identifiers into account:

```
>> assume(x > 0): abs(x), sign(x), Re(x), Im(x)
```

```
x, 1, x, 0
```

The equation `ln(z1*z2) = ln(z1) + ln(z2)` does not hold for arbitrary `z1`, `z2` in the complex plane:

```
>> expand(ln(z1*z2))
```

```
ln(z1 z2)
```

However, this identity holds if at least one of the numbers is real and positive:

```
>> assume(z1 > 0): expand(ln(z1*z2))
```

```
ln(z1) + ln(z2)
```

```
>> unassume(x): unassume(z1):
```

Example 8. If a combination of properties cannot be represented explicitly, `assume` may attach a weaker property to the identifier. In this example, the property "a prime number or the negative of a prime number" is generalized to the property "integer unequal to zero":

```
>> assume(x, Type::Prime):
   assume(x, -Type::Prime, _or)

                 Type::Integer and not Type::Zero

>> unassume(x):
```

Background:

- `assume` is an exported function of the library `property`.

`asympt` – compute an asymptotic series expansion

`asympt(f, x)` computes the first terms of an asymptotic series expansion of `f` with respect to the variable `x` around the point `infinity`.

Call(s):

- `asympt(f, x)`
- `asympt(f, x <= x0> <, order> <, dir>)`

Parameters:

`f`	—	an arithmetical expression representing a function in `x`
`x`	—	an identifier
`x0`	—	the expansion point: an arithmetical expression; if not specified, the default expansion point `infinity` is used
`order`	—	the number of terms to be computed: a nonnegative integer; the default order is given by the environment variable `ORDER` (default value 6)

Options:

`dir`	—	either ***`Left`*** or ***`Right`***. With ***`Left`***, the expansion is valid for real $x < x_0$; with ***`Right`***, it is valid for $x > x_0$. For finite expansion points x_0, the default is ***`Right`***.

Return Value: an object of domain type `Series::gseries` or `Series::Puiseux`, or an expression of type `"asympt"`.

Side Effects: The function is sensitive to the environment variable `ORDER`, which determines the default number of terms in series computations.

Overloadable by: `f`

Related Functions: `limit`, `O`, `ORDER`, `series`, `Series::gseries`, `Series::Puiseux`, `taylor`, `Type::Series`

Details:

- `asympt` is used to compute an asymptotic expansion of `f` when `x` tends to `x0`. If such an expansion can be computed, a series object of domain type `Series::gseries` or `Series::Puiseux` is returned.

  In contrast to the default behavior of `series`, `asympt` computes directed expansions that may be valid along the real line only.

- `asympt` can compute more general types of asymptotic expansions than the related function `series`. Cf. example 5.

  If `x0` is a regular point of `f`, a pole, or an algebraic branch point, then `asympt` returns a Puiseux expansion. In this case it is recommended to use the faster function `series` instead.

- If `asympt` cannot compute an asymptotic expansion, then a symbolic expression of type `"asympt"` is returned. Cf. example 4.

- The number of requested terms for the expansion is `order` if specified. Otherwise, the value of the environment variable `ORDER` is used. You can change the default value 6 by assigning a new value to `ORDER`.

  The number of terms is counted from the lowest degree term on for finite expansion points, and from the highest degree term on for expansions around infinity, i.e., "`order`" has to be regarded as a "relative truncation order".

  The actual number of terms in the resulting series expansion may differ from the requested number of terms. See `series` for details. NOTE

- The function `asympt` returns an object of domain type `Series::gseries` or `Series::Puiseux`. It can be manipulated via the standard arithmetic operations and various system functions. For example, `coeff` returns the coefficients; `expr` converts the series to an expression, removing the error term; `lmonomial` returns the leading monomial; `lterm` returns the leading term; `lcoeff` returns the leading coefficient; `map` applies a function to the coefficients; `nthcoeff` returns the n-th coefficient, `nthterm` the n-th term, and `nthmonomial` the n-th monomial.

Example 1. We compute an asymptotic expansion for $x \to \infty$:

```
>> s := asympt(sin(1/x + exp(-x)) - sin(1/x), x)

         1          1              1          /      1     \
      ------ - ----------- + ------------ + O| --------- |
      exp(x)      2              4            |   6       |
               2 x  exp(x)   24 x  exp(x)     \ x  exp(x) /
```

The leading term and the third term are extracted:

```
>> lmonomial(s), nthterm(s, 3)

                        1         1
                     ------, ---------
                     exp(x)   4
                             x  exp(x)
```

In the following call, only 2 terms of the expansion are requested:

```
>> asympt(
     exp(sin(1/x + exp(-exp(x)))) - exp(sin(1/x)), x, 2
   )

            1              1           /        1        \
       ----------- + ------------- + O| -------------- |
       exp(exp(x))   x exp(exp(x))    |  2              |
                                      \ x  exp(exp(x)) /

>> delete s:
```

Example 2. We compute a expansion around a finite real point. By default, the expansion is valid "to the right" of the expansion point:

```
>> asympt(abs(x/(1+x)), x = 0)

                    2    3    4    5    6      7
               x - x  + x  - x  + x  - x  + O(x )
```

A different expansion is valid "to the left" of the expansion point:

```
>> asympt(abs(x)/(1 + x), x = 0, Left)

                    2    3    4    5    6        7
             - x + x  - x  + x  - x  + x  + O(- x )
```

Example 3. The following expansion is exact. Therefore, it has no "error term":

```
>> asympt(exp(x+1/x), x = infinity)

                         / 1 \
              exp(x) exp| - |
                         \ x /
```

Example 4. Here is an example where `asympt` cannot compute an asymptotic series expansion:

```
>> asympt(cos(x*s)/s, x = infinity)

                  / cos(s x)              \
            asympt| --------, x = infinity |
                  \     s                 /
```

Example 5. If we apply the function `series` to the following expression, it essentially returns the expression itself:

```
>> series((ln(ln(x)+ln(ln(x))) - ln(ln(x))) /
          ln(ln(x)+ln(ln(ln(x))))*ln(x), x = infinity)

      ln(x) (- ln(ln(x)) + ln(ln(x) + ln(ln(x))))    /  1 \
      ------------------------------------------- + O| -- |
               ln(ln(x) + ln(ln(ln(x))))             |  6 |
                                                     \ x  /
```

In this example, `asympt` computes a more detailed series expansion:

```
>> asympt((ln(ln(x)+ln(ln(x))) - ln(ln(x))) /
          ln(ln(x)+ln(ln(ln(x))))*ln(x), x = infinity)

        ln(ln(x))    ln(ln(ln(x)))     /        1         \
     1 - --------- - --------------- + O| ---------------- |
         2 ln(x)    ln(x) ln(ln(x))    |               5 |
                                       \ ln(x) ln(ln(x))  /
```

Changes:

- `asympt` now may return an object of type `Series::Puiseux`.

`bernoulli` – the Bernoulli numbers and polynomials

`bernoulli(n)` returns the n-th Bernoulli number.

`bernoulli(n, x)` returns the n-th Bernoulli polynomial in x.

Call(s):

- `bernoulli(n)`
- `bernoulli(n, x)`

Parameters:

`n` — an arithmetical expression representing a nonnegative integer
`x` — an arithmetical expression

Return Value: an arithmetical expression.

Side Effects: When called with a floating point value `x`, the function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Details:

- The Bernoulli polynomials are defined by the generating function

$$\frac{t\,e^{x\,t}}{e^t - 1} = \sum_{n=0}^{\infty} \frac{\texttt{bernoulli}(n, x)}{n!}\, t^n \ .$$

- The Bernoulli numbers are defined by `bernoulli(n)` = `bernoulli(n,0)`.
- An error occurs if `n` is a numerical value not representing a nonnegative integer. If `n` contains non-numerical symbolic identifiers, then a symbolic call `bernoulli(n)` is returned. Various simplifications of `bernoulli(n,x)` are implemented for symbolic `n` and special numerical values of `x`. Cf. example 3.
- Note that floating point evaluation for high degree polynomials may be numerically unstable. Cf. example 4.

Example 1. The first Bernoulli numbers are:

```
>> bernoulli(n) $ n = 0..11

      1, -1/2, 1/6, 0, -1/30, 0, 1/42, 0, -1/30, 0, 5/66, 0
```

The first Bernoulli polynomials:

```
>> bernoulli(n, x) $ n = 0..4

                                           2
              2            x   3 x      3   2      3    4
 1, x - 1/2, x  - x + 1/6, - - ---- + x , x  - 2 x  + x  - 1/30
                           2    2
```

If `n` is symbolic, then a symbolic call is returned:

```
>> bernoulli(n, x), bernoulli(n + 3/2, x), bernoulli(n + 5*I, x)

  bernoulli(n, x), bernoulli(n + 3/2, x), bernoulli(n + 5 I, x)
```

An error occurs if `n` represents a numerical value that is not a nonnegative integer:

```
>> bernoulli(sin(3), x)

 Error: first argument must be symbolic or a nonnegative \
 integer [bernoulli]
```

Example 2. If `x` is not an indeterminate, then the evaluation of the Bernoulli polynomial at the point `x` is returned:

```
>> bernoulli(50, 1 + I)

           132549963452557267373179389125/66 + 25 I

>> bernoulli(3, 1 - y), expand(bernoulli(3, 1 - y))

       2                                                 2
 (1 - y)  (3 - 3 y)   (1 - y) (3 - 3 y)   y         3 y    y    3
 ------------------ - ----------------- - - + 1/2, ---- - - - y
         3                   2            2          2    2
```

Example 3. Certain simplifications occur for some special numerical value of x, even if n is symbolic:

```
>> bernoulli(n, -2), bernoulli(n, -1/2), bernoulli(n, -1/6)

     n                        n  n - 1
 (-1)  bernoulli(n, 2) + n (-1)  2     ,

                    n   1 - n                n      n - 1
    bernoulli(n) (-1)  (2      - 1) + n (-1)  (1/2)      ,

       n                           n      n - 1
    (-1)  bernoulli(n, 1/6) + n (-1)  (1/6)

>> bernoulli(n, 1/2), bernoulli(n, 2/3), bernoulli(n, 0.7)

               1 - n           n
 bernoulli(n) (2      - 1), (-1)  bernoulli(n, 1/3),

       n
    (-1)  bernoulli(n, 0.3)
```

Example 4. Float evaluation of high degree polynomials may be numerically unstable:

```
>> exact := bernoulli(50, 1 + I): float(exact);

                    2.00833278e27 + 25.0 I

>> bernoulli(50, float(1 + I))

                 2.00833278e27 + 437450444.9 I

>> DIGITS := 40: bernoulli(50, float(1 + I))

 2008332779584201020805748320.075757575758 + 25.000000000000000\
 000004355283802703612O8 I

>> delete exact, DIGITS:
```

Background:

- Reference: M. Abramowitz and I. Stegun, "Handbook of Mathematical Functions", Dover Publications Inc., New York (1965).

`besselI`, `besselJ`, `besselK`, `besselY` – the Bessel functions

`besselJ(v, z)`, `besselI(v, z)`, `besselY(v, z)`, and `besselK(v, z)` represent the Bessel functions:

$$J_v(z) = \frac{(z/2)^v}{\sqrt{\pi}\,\Gamma(v+1/2)} \int_0^{\pi} \cos(z\,\cos(t))\,\sin(t)^{2v}\,\mathrm{d}t,$$

$$I_v(z) = \frac{(z/2)^v}{\sqrt{\pi}\,\Gamma(v+1/2)} \int_0^{\pi} \exp(z\,\cos(t))\,\sin(t)^{2v}\,\mathrm{d}t,$$

$$Y_v(z) = \frac{J_v(z)\,\cos(v\,\pi) - J_{-v}(z)}{\sin(v\,\pi)}, \quad K_v(z) = \frac{\pi}{2}\,\frac{I_{-v}(z) - I_v(z)}{\sin(v\,\pi)}$$

`besselJ(v,z)` and `besselY(v,z)` are the Bessel functions of the first and second kinds, respectively; `besselI(v,z)` and `besselK(v,z)` are the corresponding modified Bessel functions.

Call(s):

- `besselI(v, z)`
- `besselJ(v, z)`
- `besselK(v, z)`
- `besselY(v, z)`

Parameters:

`v, z` — arithmetical expressions

Return Value: an arithmetical expression.

Overloadable by: `z`

Side Effects: When called with floating point arguments, these functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Details:

- The Bessel functions are defined for complex arguments v and z.

- A floating point value is returned if either of the arguments is a floating point number and the other argument is numerical. For most exact arguments the Bessel functions return an unevaluated function call. Special values at index $v = 0$ and/or argument $z = 0$ are implemented. Explicit symbolic expressions are returned, when the index v is a half integer. Cf. example 2.
- For nonnegative integer indices v some of the Bessel functions have a branch cut along the negative real axis. A jump occurs when crossing this cut. Cf. example 3.
- If floating point approximations are desired for arguments that are exact numerical expressions, then we recommend to use `besselJ(v, float(x))` rather than `float(besselJ(v, x))`. In particular, for half integer indices the symbolic result `besselJ(v,x)` is costly to compute. Further, floating point evaluation of the resulting symbolic expression may be numerically unstable. Cf. example 4.

Example 1. Unevaluated calls are returned for exact or symbolic arguments:

```
>> besselJ(2, 1 + I), besselK(0, x), besselY(v, x)

         besselJ(2, 1 + I), besselK(0, x), besselY(v, x)
```

Floating point values are returned for floating point arguments:

```
>> besselI(2, 5.0), besselK(3.2 + I, 10000.0)

      17.50561497, 1.423757712e-4345 + 4.555796986e-4349 I
```

Example 2. Bessel functions can be expressed in terms of elementary functions if the index is an odd integer multiple of 1/2:

```
>> besselJ(1/2, x), besselY(3/2, x)

                        1/2 /            cos(x) \
                  1/2  2    | - sin(x) - ------ |
         sin(x) 2           \               x   /
         -----------, --------------------------
           1/2   1/2            1/2   1/2
          x    PI              x    PI

>> besselI(7/2, x), besselK(-7/2, x)
```

```
  1/2 /         / 15      \           / 6   15 \ \
 2    | cosh(x) | -- + 1 | - sinh(x) | - + -- | |
      |         |  2     |           | x    3 | |
      \         \ x      /           \     x  / /
 ------------------------------------------------,
                   1/2   1/2
                  x    PI

     1/2  1/2         / 6   15   15     \
   PI    2    exp(-x) | - + -- + -- + 1 |
                      | x    2    3     |
                      \     x    x      /
   --------------------------------------
                        1/2
                     2 x
```

Example 3. The negative real axis is a branch cut of the Bessel functions for non-integer indices v. A jump occurs when crossing this cut:

```
>> besselI(-3/4, -1.2), besselI(-3/4, -1.2 + I/10^10),
   besselI(-3/4, -1.2 - I/10^10)

 - 0.76061492 - 0.76061492 I, - 0.76061492 - 0.7606149199 I,

    - 0.76061492 + 0.7606149199 I
```

Example 4. The symbolic expressions returned by Bessel functions with half integer indices may be unsuitable for floating point evaluation:

```
>> y := besselJ(51/2, PI)

 /  1/2 / 450675225   52650   1466947857375           \ \
 | 2    | --------- - ----- - ------------- + ... + 1 | | / PI
 |      |      4        2           6                 | |
 \      \    PI       PI          PI                  / /
```

Floating point evaluation of this exact result is subject to numerical cancellation. The following result is dominated by round-off:

```
>> float(y)

                             8.862488737
```

The numerical working precision has to be increased to obtain a more accurate result:

```
>> DIGITS:= 39: float(y)

  0.000000000000000000001160130057519777842731692376779416479 77
```

Direct floating point evaluation via the Bessel function yields a correct result within working precision:

```
>> DIGITS := 5: besselJ(51/2, float(PI))

                           1.1601e-21

>> delete y, DIGITS:
```

Example 5. The functions `diff`, `float`, `limit`, and `series` handle expressions involving the Bessel functions:

```
>> diff(besselJ(0, x), x, x), float(ln(3 + besselI(17, sqrt(PI))))

           besselJ(1, x)
           ------------- - besselJ(0, x), 1.098612289
                 x

>> limit(besselJ(2, x^2 + 1)*sqrt(x), x = infinity)

                                 0

>> series(besselY(3, x)/x, x = infinity, 3)

  1/2    /     7 PI \        1/2    /     7 PI \
 2    sin| x - ---- |   35 2    cos| x - ---- |
         \       4  /               \       4  /
 -------------------- + ---------------------- -
      3/2   1/2                    5/2   1/2
     x    PI                    8 x    PI

        1/2    /     7 PI \
    945 2    sin| x - ---- |
               \       4  /    /   1  \
    ----------------------- + O| ---- |
             7/2   1/2         |  9/2 |
        128 x    PI            \ x    /
```

Background:

- The Bessel functions are regular (holomorphic) functions of z throughout the z-plane cut along the negative real axis, and for fixed $z \neq 0$, each is an entire (integral) function of v.

- $J_v(z)$ and $Y_v(z)$ satisfy Bessel's equation in $w(v, z)$:

$$z^2 \frac{d^2w}{dz^2} + z \frac{dw}{dz} + (z^2 - v^2)\, w = 0.$$

  Correspondingly, $I_v(z)$ and $K_v(z)$ satisfy the modified Bessel equation:

$$z^2 \frac{d^2w}{dz^2} + z \frac{dw}{dz} - (z^2 + v^2)\, w = 0.$$

- When the index v is an integer, the Bessel functions are governed by reflection formulas:

$$I_{-v}(z) = I_v(z),\ J_{-v}(z) = (-1)^v J_v(z),$$

$$K_{-v}(z) = K_v(z),\ Y_{-v}(z) = (-1)^v Y_v(z).$$

`beta` – the beta function

`beta(x, y)` represents the beta function $\Gamma(x)\,\Gamma(y)/\Gamma(x+y)$.

Call(s):

- `beta(x, y)`

Parameters:

`x, y` — arithmetical expressions or floating point intervals

Return Value: an arithmetical expression or a floating point interval.

Overloadable by: `x`

Side Effects: When called with floating point arguments, the function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Related Functions: `binomial`, `fact`, `gamma`, `psi`

Details:

- The beta function is defined for complex arguments x and y.
- The result is expressed by calls to the gamma function if both arguments are of type `Type::Numeric`. Note that the beta function may have a regular value, even if $\Gamma(x)$ or $\Gamma(y)$ and $\Gamma(x+y)$ are singular. In such cases `beta` returns the limit of the quotients of the singular terms.
- A floating point value is returned if both arguments are numerical and at least one of them is a floating point value.
- An unevaluated call of `beta` is returned, if none of the arguments vanishes and at least one of the arguments does not evaluate to a number of type `Type::Numeric`.

Example 1. We demonstrate some calls with exact and symbolic input data:

```
>> beta(1, 5), beta(I, 3/2), beta(1, y + 1), beta(x, y)

                    1/2
                  PI    gamma(I)     1
           1/5, ----------------, -----, beta(x, y)
                 2 gamma(3/2 + I)  y + 1
```

Floating point values are computed for floating point arguments:

```
>> beta(3.5, sqrt(2)), beta(sqrt(2), 2.0 + 10.0*I)

         0.1395855454, - 0.01112350756 - 0.03108193098 I
```

Example 2. The gamma function is singular if its argument is a nonpositive integer. Nevertheless, `beta` has a regular value for the following arguments:

```
>> beta(-3, 2)

                               1/6
```

Example 3. The functions `diff`, `expand` and `float` handle expressions involving `beta`:

```
>> diff(beta(x^2, x), x)

            2                      2             2
   beta(x, x ) (psi(x) + 2 x psi(x ) - psi(x + x ) (2 x + 1))
```

```
>> expand(beta(x - 1, y + 1))

                          y gamma(x) gamma(y)
                         --------------------
                         gamma(x + y) (x - 1)

>> float(beta(100, 1000))

                           7.730325902e-147
```

Example 4. The functions `diff` and `series` can handle `beta`:

```
>> diff(beta(x, y), x);
   diff(beta(x, y), y);

             beta(x, y) (psi(x) - psi(x + y))

             beta(x, y) (psi(y) - psi(x + y))

>> normal(series(beta(x, y), y = 0, 3))

                         /                             2
 1                       |                psi(x, 1)   PI
 - - (EULER + psi(x)) + y | EULER psi(x) - --------- + --- +
 y                       \                    2        12

        2          2 \
   EULER    psi(x)   |      2
   ------ + ------- | + O(y )
     2         2    /

>> series(beta(x, x), x = infinity, 4)

        1/2          1/2             1/2
     2 PI          PI              PI          /     1     \
    --------- + ----------- + ------------ + O| --------- |
     1/2  2 x      3/2  2 x       5/2  2 x     |  7/2  2 x |
    x    2      4 x    2      64 x    2       \ x    2    /
```

Changes:

- Series expansions can now be computed via `series`.
- The function `rewrite` can now express `beta` in terms of `gamma` or `fact`.

`binomial` – binomial coefficients

`binomial(n, k)` represents the binomial coefficient $\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$.

Call(s):

- `binomial(n, k)`

Parameters:

`n, k` — arithmetical expressions

Return Value: an arithmetical expression.

Side Effects: When called with floating point arguments, the function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Related Functions: `beta`, `fact`, `gamma`, `psi`

Details:

- Binomial coefficients are defined for complex arguments via the `gamma` function:
$$\binom{n}{k} = \frac{\Gamma(n+1)}{\Gamma(k+1)\Gamma(n-k+1)}.$$
With $\Gamma(n+1) = n!$, this coincides with the usual binomial coefficients for integer arguments satisfying $0 \le k \le n$.

- A symbolic function call is returned if one of the arguments cannot be evaluated to a number of type `Type::Numeric`. However, for $k = 0$, $k = 1$, $k = n - 1$, and $k = n$, simplified results are returned for any n.

- Let n be a number of type `Type::Numerical`. If k evaluates to a nonnegative integer, then $n \times (n-1) \times \cdots \times (n-k+1)/k!$ is returned. If k evaluates to a negative integer, then 0 is returned. If k evaluates to a floating point number, then a floating point value is returned. In all other cases, a symbolic call of `binomial` is returned.

- A floating point value is returned if both arguments are numerical and at least one of them is a floating point value.

Example 1. We demonstrate some calls with exact and symbolic input data:

```
>> binomial(10, k) $ k=-2..12

    0, 0, 1, 10, 45, 120, 210, 252, 210, 120, 45, 10, 1, 0, 0

>> binomial(-23/12, 3), binomial(1 + I, 3)

                     -37835/10368, -1/3 I

>> binomial(n, k), binomial(n, 1), binomial(n, 4)

                binomial(n, k), n, binomial(n, 4)
```

Floating point values are computed for floating point arguments:

```
>> binomial(-235/123, 3.0), binomial(3.0, 1 + I)

            -3.624343742, 4.411293492 + 2.205646746 I
```

Example 2. The `expand` function handles expressions involving `binomial`:

```
>> binomial(n, 3) = expand(binomial(n, 3))

                                         2    3
                                     n   n    n
                   binomial(n, 3) = - - -- + --
                                     3    2    6

>> binomial(2, k) = expand(binomial(2, k))

                                     2 sin(k PI)
               binomial(2, k) = --------------------
                                k PI (1 - k) (2 - k)
```

The `float` attribute handles `binomial` if all arguments can be converted to floating point numbers:

```
>> binomial(sin(3), 5/4), float(binomial(sin(3), 5/4))

               binomial(sin(3), 5/4), -0.08360571366
```

Example 3. The functions `diff` and `series` can handle `binomial`:

```
>> diff(binomial(n, k), n);
   diff(binomial(n, k), k);

          binomial(n, k) (psi(n + 1) - psi(n - k + 1))

          binomial(n, k) (psi(n - k + 1) - psi(k + 1))

>> normal(series(binomial(n, k), k = 0, 3))

                               2
 1 + k (EULER + psi(n + 1)) + k

    /     2        2
    |   PI    EULER                      psi(n + 1, 1)
    | - --- + ------ + EULER psi(n + 1) - ------------- +
    \    12      2                              2

              2 \
    psi(n + 1)  |      3
    ----------- | + O(k )
         2      /

>> series(binomial(2*n, n), n = infinity, 4)

          2 n           2 n             2 n         /  2 n \
         2             2               2            | 2    |
     ---------- - ------------ + -------------- + O| ---- |
      1/2   1/2       3/2   1/2        5/2   1/2    |  7/2 |
     n    PI      8 n    PI      128 n    PI       \ n    /
```

Changes:

- The functions `diff` and `series` were overloaded for `binomial`.

`bool` – Boolean evaluation

`bool(b)` evaluates the Boolean expression `b`.

Call(s):

- `bool(b)`

Parameters:

`b` — a Boolean expression

Return Value: `TRUE`, `FALSE`, or `UNKNOWN`.

Overloadable by: `b`

Related Functions: `_lazy_and`, `_lazy_or`, `FALSE`, `if`, `is`, `repeat`, `TRUE`, `UNKNOWN`, `while`

Details:

- The function `bool` serves for reducing Boolean expressions to one of the Boolean constants `TRUE`, `FALSE`, or `UNKNOWN`.

  Boolean expressions are expressions that are composed of equalities, inequalities, elementhood relations, and these constants, combined via the logical operators `and`, `or`, `not`.

  The function `bool` evaluates all equalities and inequalities inside a Boolean expression to either `TRUE` or `FALSE`. The resulting logical combination of the Boolean constants is reduced according to the rules of MuPAD's three state logic (see `and`, `or`, `not`).

- Equations `x = y` and inequalities `x <> y` are evaluated *syntactically* by `bool`. It does not test equality in any mathematical sense.

- Inequalities `x < y`, `x <= y` etc. can be evaluated by `bool` if and only if `x` and `y` are real numbers of type `Type::Real`. Otherwise, an error occurs.

- `bool` evaluates *all* subexpressions of a Boolean expression before simplifying the result. The functions `_lazy_and`, `_lazy_or` provide an alternative: "lazy Boolean evaluation".

- There is no need to use `bool` in the conditional part of `if`, `repeat`, and `while` statements. Internally, these statements enforce Boolean evaluation by `_lazy_and` and `_lazy_or`. Cf. example 5.

- Use `simplify` with the option `logic` to simplify expressions involving symbolic Boolean subexpressions. Cf. example 7.

- `bool` is overloadable not only for domains, but also for function environments. This means that, if `f` evaluates to a function environment, then `bool(f(x1, ..., xn))` returns `f::bool(x1, ..., xn)`, or an error if no slot `f::bool` exists.

- `bool` is a function of the system kernel.

Example 1. MuPAD realizes that 1 is less than 2:

```
>> 1 < 2 = bool(1 < 2)
```

```
(1 < 2) = TRUE
```

Note that `bool` can only compare real numbers of syntactical type `Type::Real`:

```
>> bool(PI < 2 + sqrt(2))
```

```
Error: Can't evaluate to boolean [_less]
```

One can compare floating point approximations. Alternatively, one can use `is`:

```
>> bool(float(PI) < float(2 + sqrt(2))), is(PI < 2 + sqrt(2))
```

```
TRUE, TRUE
```

Example 2. The Boolean operators `and`, `or`, `not` do not evaluate equations and inequalities logically, and return a symbolic Boolean expression. Boolean evaluation and simplification is enforced by `bool`:

```
>> a = a and 3 < 4
```

```
a = a and 3 < 4
```

```
>> bool(a = a and 3 < 4)
```

```
TRUE
```

Example 3. `bool` handles the special Boolean constant `UNKNOWN`:

```
>> bool(UNKNOWN and 1 < 2), bool(UNKNOWN or 1 < 2),
   bool(UNKNOWN and 1 > 2), bool(UNKNOWN or 1 > 2)
```

```
UNKNOWN, TRUE, FALSE, UNKNOWN
```

Example 4. `bool` must be able to reduce all parts of a composite Boolean expression to one of the Boolean constants. No symbolic Boolean subexpressions may be involved:

```
>> b := b1 and b2 or b3: bool(b)
```

```
Error: Can't evaluate to boolean [bool]
```

```
>> b1 := 1 < 2: b2 := x = x: b3 := FALSE: bool(b)
```

```
TRUE
```

```
>> delete b, b1, b2, b3:
```

Example 5. There is no need to use `bool` explicitly in the conditional parts of `if`, `repeat`, and `while` statements. Note, however, that these structures internally use "lazy evaluation" via `_lazy_and` and `_lazy_or` rather than "complete Boolean evaluation" via `bool`:

```
>> x := 0: if x <> 0 and sin(1/x) = 0 then 1 else 2 end
```

```
2
```

In contrast to "lazy evaluation", `bool` evaluates *all* conditions. Consequently, a division by zero occurs in the evaluation of `sin(1/x) = 0`:

```
>> bool(x <> 0 and sin(1/x) = 0)
```

```
Error: Division by zero
```

```
>> delete x:
```

Example 6. Note that `bool` does not operate recursively. The following calls are completely different, the first one comparing the expression `TRUE = TRUE` and the constant `TRUE` (syntactically), the second one comparing the result of another `bool`-call with `TRUE`:

```
>> bool((TRUE = TRUE) = TRUE);
   bool(bool(TRUE = TRUE) = TRUE)
```

```
FALSE

TRUE
```

Since `if`, `while` and similar constructs use the same Boolean evaluation internally, this also effects conditions in such clauses:

```
>> if (is(a < b) = TRUE) or (3 = 3) then YES else NO end;
   if (is(a < b) or (3 = 3)) = TRUE then YES else NO end
```

```
YES

NO
```

Example 7. Expressions involving symbolic Boolean subexpressions cannot be processed by `bool`. However, `simplify` with the option `logic` can be used for simplification:

```
>> (b1 and b2) or (b1 and (not b2)) and (1 < 2)
```

```
b1 and b2 or b1 and not b2 and 1 < 2
```

```
>> simplify(%, logic)
```

```
b1
```

Changes:

- `bool` is now also overloadable by function environments.

`break` – terminate a loop or a `case` switch prematurely

`break` terminates `for`, `repeat`, `while` loops, and `case` statements.

Call(s):

- `break`
- `_break()`

Related Functions: `case`, `for`, `next`, `quit`, `repeat`, `return`, `while`

Details:

- The `break` statement is equivalent to the function call `_break()`. The return value is the void object of type `DOM_NULL`.
- Inside `for`, `repeat`, `while`, and `case` statements, the `break` statement exits from the loop/switch. Execution proceeds with the next statement after the `end` clause of the loop/switch.
- In nested loops, only the innermost loop is terminated by `break`.
- `break` also terminates a statement sequence `_stmtseq(..., break, ...)`.
- Outside `for`, `repeat`, `while`, `case`, and `_stmtseq`, the `break` statement has no effect.
- `_break` is a function of the system kernel.

Example 1. Loops are exited prematurely by `break`:

```
>> for i from 1 to 10 do
     print(i);
     if i = 2 then break end_if
   end_for
```

```
1

2
```

```
>> delete i:
```

Example 2. In a `case` statement, all commands starting with the first matching branch are executed:

```
>> x := 2:
   case x
    of 1 do print(1); x^2;
    of 2 do print(2); x^2;
    of 3 do print(3); x^2;
    otherwise print(UNKNOWN)
   end_case:

                                 2

                                 3

                              UNKNOWN
```

In the next version, `break` ensures that only the statements in the matching branch are evaluated:

```
>> case x
    of 1 do print(1); x^2; break;
    of 2 do print(2); x^2; break;
    of 3 do print(3); x^2; break;
    otherwise print(UNKNOWN)
   end_case:

                                 2

>> delete x:
```

`builtin` – representatives of C-functions of the MuPAD kernel

`builtin` represents a C-function of the system kernel.

Call(s):

- `builtin(i, j, str, tbl)`
- `builtin(i, j1, str1, str)`

Parameters:

`i`	—	a number corresponding to a C-function of the kernel: a nonnegative integer
`j`	—	a number corresponding to a C-function of the kernel: a nonnegative integer or `NIL`
`str`	—	the name of the created `DOM_EXEC` object: a character string
`tbl`	—	the remember table of the function: a table or `NIL`
`j1`	—	the precedence of an operator: a nonnegative integer
`str1`	—	the operator symbol: a character string or `NIL`

Return Value: an object of type `DOM_EXEC`.

Related Functions: `funcenv`

Details:

- `builtin` is only intended for internal use! A user is not supposed to call this low-level function. NOTE
- The function `builtin` provides an interface between the MuPAD language and the C-functions of the MuPAD kernel. The MuPAD functions returned by `builtin` are elements of the basic type `DOM_EXEC`. They may only be used as first or second entry of `function environments` created by `funcenv`.
- Functions used as the first argument in `funcenv` serve for evaluating function calls of the function environment. A kernel function serving this purpose must be produced by a call `builtin(i, j, str, tbl)`. The string `str` is used for the ouput of symbolic calls of the kernel function. The table `tbl` is the remember table. Cf. example 2. If `NIL` is used, no remember table is associated with the function.
- Functions used as the second argument in `funcenv` determine the output of symbolic function calls. A kernel function serving this purpose must be produced by a call `builtin(i, j1, str1, str)`. The number `j1` defines the output priority of the function. If symbolic function calls are to be presented in operator notation, the string `str1` is used as the operator symbol. Cf. example 3. `NIL` must be used if the function does not represent an operator. The string `str` is used for the output of the `DOM_EXEC` object itself.
- `builtin` is a function of the system kernel.

Example 1. The operands of a function environment such as `_mult` can be viewed by `expose`. The following two kernel functions are in charge of evaluating products and displaying the result on the screen, respectively:

```
>> expose(op(_mult, 1)), expose(op(_mult, 2))
```

```
 builtin(815, NIL, "_mult", NIL),

    builtin(1100, 1100, "*", "_mult")
```

```
>> _mult(a, b) = builtin(815, NIL, "_mult", NIL)(a, b)
```

                                        a b = a b

Example 2. We demonstrate that it is possible to manipulate the remember table of kernel functions. The function environment `isprime` uses a C-function of the kernel to evaluate its argument:

```
>> expose(isprime)
```

```
                 builtin(1000, 1305, "isprime", NIL)
```

It does not regard 1 as a prime number:

```
>> isprime(1)
```

```
                                FALSE
```

We unprotect the system function and associate the value `TRUE` with the call `isprime(1)`:

```
>> unprotect(isprime): isprime(1) := TRUE:
```

The value is stored in the remember table. This is the fourth entry of the builtin function evaluating the arguments of `isprime`:

```
>> expose(isprime)
```

```
                 /                            table(      \
          builtin| 1000, 1305, "isprime",       1 = TRUE  |
                 \                            )           /
```

After this modification, `isprime` regards 1 as a prime number:

```
>> isprime(1)
```

```
                                TRUE
```

We restore the original behavior of `isprime` by substituting the original value `NIL` of the remember table:

```
>> isprime := subsop(isprime, [1, 4] = NIL): protect(isprime):
```

```
>> isprime(1)
```

```
                                FALSE
```

Example 3. We demonstrate how the output symbol of the kernel function `_power` can be changed. This function is in charge of representing powers:

```
>> op(a^b, 0), _power(a, b)

                                 b
                         _power, a
```

The second operand of the function environment `_power` is the builtin function that determines the output:

```
>> expose(op(_power,2))

             builtin(1100, 1200, "^", "_power")
```

The third operand of this object is the symbol that is used for representing symbolic powers. We want to replace it by `**`. However, since the system function `_power` is protected, we have to apply `unprotect` before we can modify the function environment:

```
>> unprotect(_power): _power := subsop(_power, [2, 3] = "**"):

>> expose(op(_power,2)), a^b

          builtin(1100, 1200, "**", "_power"), a**b
```

We restore the original behavior of `_power`:

```
>> _power := subsop(_power, [2, 3] = "^"):  protect(_power):
```

`bytes` – the memory used by the current MuPAD session

`bytes()` returns the current memory consumption.

Call(s):

- `bytes()`

Return Value: a sequence of three integers.

Related Functions: `rtime`, `share`, `time`

Details:

- `bytes` returns the following three numbers:
  - The number of bytes used logically; this is the amount of memory which is actually used for storing MuPAD data.
  - The number of bytes physically allocated by the memory management; this is the amount of memory MuPAD has allocated from the operating system. The difference between the physical and the logical bytes is the amount of memory which has already been reserved for future calculations.
  - On computers with a virtual memory, the third number is the constant $2^{31} - 1$. On other computers such as the Apple Macintosh, the remaining free space (on the *program heap*) is returned.
- `bytes` is a function of the system kernel.

Example 1. In a freshly started MuPAD session, `bytes` may return the following data on the memory consumption of the session:

```
>> bytes()

                    506584, 717312, 2147483647
```

Each computation increases the memory usage:

```
>> int(x, x): bytes()

                   2040956, 2201624, 2147483647
```

`card` – the cardinality of a set

`card(set)` returns the cardinality of `set`.

Call(s):

- `card(set)`
- `card(d)`

Parameters:

`set` — a set of type `DOM_SET`, or a set-theoretic expression
`d` — a domain representing a set

Overloadable by: `set, d`

Return Value: a nonnegative integer, or `infinity`.

Related Functions: `nops`

Details:

- If `set` is a `DOM_SET`, the number of operands is returned; `card` does not attempt to investigate whether the members of `set` really represent pairwise different mathematical objects.
- `card` does not distinguish different infinite cardinals; it just returns `infinity` if `set` is infinite.
- `card` returns a symbolic call to itself if it cannot determine the cardinality.
- If applied to a domain `d`, `card` returns the domain entry `d::size`. A domain that does not have this entry is not regarded as a set.

Example 1. The cardinality of a finite set equals the number of its operands:

```
>> card({1, 2, 3})

                                  3
```

This holds true even if there exist two operands of the set that represent the same mathematical object:

```
>> card({1, 1.0})

                                  2
```

Example 2. `card` does not distinguish different sizes of infinite sets:

```
>> card(R_), card(Z_)

                          infinity, infinity
```

Example 3. Set–theoretic expressions containing symbols are legal input, but usually `card` will not be able to determine their cardinality:

```
>> card(S union {3})

                          card({3} union S)
```

Example 4. Domains that have a "size" entry are regarded as sets:

```
>> card(Dom::IntegerMod(7))

                    7
```

Changes:

- card is a new function.

---

case – switch statement

case-end_case statement allows to switch between various branches in a program.

Call(s):

- ```
  case x
    of match1 do statements1
    of match2 do statements2
    ...
    <otherwise otherstatements>
  end_case
  ```
- ```
  _case(x, match1, statements1, match2, statements2, ...
        <, otherstatements>)
  ```

Parameters:

x, match1, match2, ...	— arbitrary MuPAD objects
statements1, ..., otherstatements	— arbitrary sequences of statements

Return Value: the result of the last command executed inside the case statement. The void object of type DOM_NULL is returned if no matching branch was found and no otherwise branch exists. NIL is returned if a matching branch was encountered, but no command was executed inside this branch.

Related Functions: break, if, return

Details:

- The case statement is a control structure that extends the functionality of the if statement. In a case statement, an object is compared with a number of given values and one or more statement sequences are executed.

- If the value of `x` equals one of the values `match1`, `match2` etc., the first matching branch *and all its following branches (including `otherwise`)* are executed, until the execution is terminated by a `break` or a `return` statement, or the `end_case`.
- If the value of `x` does not equal any of the values `match1, match2, ...`, only the `otherwise` branch is executed. If no `otherwise` branch exists, the `case` statement terminates and returns the void object of type `DOM_NULL`.
- The keyword `end_case` may be replaced by the keyword `end`.
- `_case` is a function of the system kernel.

Example 1. All statements after the first match are executed:

```
>> x := 2:
   case x
     of 1 do print(1)
     of 2 do print(4)
     of 3 do print(9)
     otherwise print("otherwise")
   end_case:

                                4

                                9

                           "otherwise"
```

`break` may be used to ensure that only one matching branch is executed:

```
>> case x
     of 1 do print(1); 1; break
     of 2 do print(4); 4; break
     of 3 do print(9); 9; break
     otherwise print("otherwise")
   end_case:

                                4

>> delete x:
```

Example 2. The functionality of the `case` statement allows to share code that is to be used in several branches. The following function uses the statement `print(x, "is a real number")` for the three branches that correspond to real MuPAD numbers:

```
>> isReal := proc(x)
   begin
      case domtype(x)
        of DOM_INT do
        of DOM_RAT do
        of DOM_FLOAT do print(x, "is a real number"); break
        of DOM_COMPLEX do print(x, "is not a real number"); break
        otherwise print(x, "cannot decide");
      end_case
   end_proc:
   isReal(3), isReal(3/7), isReal(1.23), isReal(3 + I), isReal(z)

                     3, "is a real number"

                    3/7, "is a real number"

                   1.23, "is a real number"

                 3 + I, "is not a real number"

                      z, "cannot decide"

>> delete isReal:
```

Example 3. The correspondence between the functional and the imperative form of the `case` statement is demonstrated:

```
>> hold(_case(x, match1, (1; break), match2, (4; break),
              print("otherwise")))

                     case x
                       of match1 do
                         1;
                         break
                       of match2 do
                         4;
                         break
                        otherwise
                         print("otherwise")
                     end_case

>> hold(_case(x, match1, (1; break), match2, (4; break)))

                         case x
                           of match1 do
                             1;
```

```
        break
      of match2 do
        4;
        break
    end_case
```

Background:

- The functionality of the `case` statement corresponds to the `switch` statement of the C programming language.

`ceil, floor, round, trunc` – rounding to an integer

`ceil` rounds a number to the next larger integer.

`floor` rounds a number to the next smaller integer.

`round` rounds a number to the nearest integer.

`trunc` rounds a number to the next integer in the direction of 0.

Call(s):

- `ceil(x)`
- `floor(x)`
- `round(x)`
- `trunc(x)`

Parameters:

`x` — an arithmetical expression or a floating point interval

Return Value: an arithmetical expression.

Overloadable by: `x`

Side Effects: The functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Related Functions: `frac`

Details:

- For complex arguments, rounding is applied separately to the real and the imaginary parts.
- Integers are returned for real numbers, floating point intervals and exact expressions representing real numbers. Unevaluated function calls are returned for arguments that contain symbolic identifiers.
- If you think of `x` as a floating point number, then `trunc(x)` truncates the digits after the decimal point. Thus, `trunc` coincides with `floor` for real positive arguments and with `ceil` for real negative arguments, respectively.
- If the argument is a floating point number of absolute value larger than $10^{\texttt{DIGITS}}$, the resulting integer is affected by internal non-significant digits! Cf. example 2.
- Internally, exact numerical expressions that are neither integers nor rational numbers are approximated by floating point numbers before rounding. Thus, the resulting integer may depend on the present value of `DIGITS`! Cf. example 3.
- For floating point intervals, `ceil` and `floor` return the smallest (largest) integer larger (smaller) than all numbers in the interval. As for intervals with infinite borders, see example 4. `trunc` returns `0` for intervals containing 0, otherwise it behaves as described above for floating point values. `round`, when applied to a floating point interval, gives the integer closest to the midpoint of the interval. Again, see example 4 for the behavior concerning infinities.

Example 1. We demonstrate the rounding of real and complex numbers:

```
>> ceil(3.5), floor(3.5), round(3.5), trunc(3.5)

                           4, 3, 4, 3

>> ceil(-7/2), floor(-7/2), round(-7/2), trunc(-7/2)

                         -3, -4, -3, -3

>> ceil(3 + 5/2*I), floor(4.3 + 7*I), round(I/2), trunc(I/2)

                     3 + 3 I, 4 + 7 I, I, 0
```

Also symbolic expressions representing numbers can be rounded:

```
>> x := PI*I + 7*sin(exp(2)): ceil(x), floor(x), round(x), trunc(x)
```

```
             7 + 4 I, 6 + 3 I, 6 + 3 I, 6 + 3 I
```

Rounding of expressions with symbolic identifiers produces unevaluated function calls:

```
>> delete x: ceil(x), floor(x - 1), round(x + 1), trunc(x^2 + 3)

                                                       2
       ceil(x), floor(x - 1), round(x + 1), trunc(x   + 3)
```

Example 2. Care should be taken when rounding floating point numbers of large absolute value:

```
>> x := 10^30/3.0

                          3.333333333e29
```

Note that only the first 10 decimal digits are "significant". Further digits are subject to round-off effects caused by the internal binary representation. These "insignificant" digits are part of the integer produced by rounding:

```
>> floor(x), ceil(x)

 333333333333333333307205615616, 333333333333333333307205615616

>> delete x:
```

Example 3. Exact numerical expressions are internally converted to floating point numbers before rounding. Consequently, the present setting of DIGITS can affect the result:

```
>> x := 10^30 - exp(30)^ln(10)

                                                      ln(10)
         1000000000000000000000000000000 - exp(30)
```

Note that the exact value of this number is 0. Floating point evaluation is subject to severe cancellations:

```
>> DIGITS := 10: float(x), floor(x), ceil(x)

         1.030792151e13, 10307921510400, 10307921510400
```

The floating point result is more accurate when a higher precision is used. The rounded values change accordingly:

```
>> DIGITS := 20: float(x), floor(x), ceil(x)
```

```
2896.0, 2896, 2896
```

```
>> DIGITS := 30: float(x), floor(x), ceil(x)
```

```
0.000000879168510437011718750, 0, 1
```

```
>> delete x, DIGITS:
```

Example 4. On floating point intervals, `ceil` and `floor` behave as expected:

```
>> ceil(3.5...6.7); floor(3.5...6.7)
```

```
7

3
```

Because there are finite numbers represented as `RD_INF` and `RD_NINF`, respectively, `ceil` and `floor` return very small or large representable integer in certain cases:

```
>> x := ceil(RD_NINF...RD_NINF):
   domtype(x);
   log(10, float(abs(x)))
```

```
DOM_INT

631266.8246
```

This may take quite some while (because the corresponding 630 000-digit integer must be constructed first) and output of this large integer would take even longer.

`trunc` behaves almost identical to its behavior on floats: For intervals not containing zero it is, depending on the sign, equivalent to `floor` or `ceil`; for intervals containing zero, it returns `0`:

```
>> trunc(-3.5...-2.7), trunc(-2.4...1.9), trunc(4.5...infinity)
```

```
-2, 0, 4
```

`round` returns the integer closest to the midpoint of the interval:

```
>> round(-3.5...-2.7), round(-2.4...1.9), round(4.5...infinity)
```

```
-3, 0, infinity
```

Changes:

- Floating point intervals are now handled.

---

coeff – the coefficients of a polynomial

`coeff(p)` returns a sequence of all nonzero coefficients of the polynomial `p`.

`coeff(p, x, n)` regards `p` as a univariate polynomial in `x` and returns the coefficient of the term `x^n`.

Call(s):

- `coeff(p)`
- `coeff(p, <x,> n)`
- `coeff(f <, vars>)`
- `coeff(f, <vars,> <x,> n)`

Parameters:

`p`	—	a polynomial of type `DOM_POLY`
`x`	—	an indeterminate
`n`	—	the power: a nonnegative integer
`f`	—	a polynomial expression
`vars`	—	a list of indeterminates of the polynomial: typically, identifiers or indexed identifiers

Return Value: one or more coefficients of the coefficient ring of the polynomial, or a polynomial, or `FAIL`.

Overloadable by: `p, f`

Related Functions: `collect`, `content`, `degree`, `degreevec`, `ground`, `icontent`, `lcoeff`, `ldegree`, `lmonomial`, `lterm`, `nterms`, `nthcoeff`, `nthmonomial`, `nthterm`, `poly`, `poly2list`, `tcoeff`

Details:

- If the first argument `f` is not element of a polynomial domain, then `coeff` converts the expression internally to a polynomial of type `DOM_POLY` via `poly(f)`. If a list of indeterminates is specified, the polynomial `poly(f, vars)` is considered.

  Coefficients of polynomial expressions `f` are returned as arithmetical expressions.

- There are various ways to call `coeff` with a polynomial `p` of type `DOM_POLY`:

  - `coeff(p)` returns a sequence of all nonzero coefficients of `p`. They are ordered according to the lexicographical term ordering.

    The returned coefficients are elements of the coefficient ring of `p`.

  - `coeff(p, x, n)` regards `p` as a univariate polynomial in the variable `x` and returns the coefficient of the term `x^n`.

    For univariate polynomials, the returned coefficients are elements of the coefficient ring of `p`.

    For multivariate polynomials, the coefficients are returned as polynomials of type `DOM_POLY` in the "remaining" variables.

  - `coeff(p, n)` is equivalent to `coeff(p, x, n)`, where `x` is the "main variable" of `p`. This variable is the first element of the list of indeterminates `op(p, 2)`.

- `coeff` returns 0 or a zero polynomial if the polynomial does not contain a term corresponding to the specified power `n`. In particular, this happens if `n` is larger than the degree of the polynomial.

- `coeff` returns `FAIL` if an expression cannot be regarded as a polynomial.

- The result of `coeff` is not fully evaluated. Evaluation can be enforced by the function `eval`. Cf. example 5.

- `coeff` is a function of the system kernel.

Example 1. `coeff(f)` returns a sequence of all nonzero coefficients:

```
>> f := 10*x^10 + 5*x^5 + 2*x^2: coeff(f)

                          10, 5, 2
```

`coeff(f, i)` returns a single coefficient:

```
>> coeff(f, i) $ i = 0..15

      0, 0, 2, 0, 0, 5, 0, 0, 0, 0, 10, 0, 0, 0, 0, 0

>> delete f:
```

Example 2. We demonstrate how the indeterminates influence the result:

```
>> f := 3*x^3 + x^2*y^2 + 17*x + 23*y + 2

                                   3    2  2
                 17 x + 23 y + 3 x  + x  y  + 2

>> coeff(f); coeff(f, [x, y]); coeff(f, [y, x])

                         1, 23, 3, 17, 2

                         3, 1, 17, 23, 2

                         1, 23, 3, 17, 2

>> delete f:
```

Example 3. The coefficients of `f` are selected with respect to the main variable `x` which is the first entry of the list of indeterminates:

```
>> f := 3*x^3 + x^2*y^2 + 2: coeff(f, [x, y], i) $ i = 0..3

                                  2
                           2, 0, y , 3
```

The coefficients of `f` can be selected with respect to another main variable (in this case, `y`):

```
>> coeff(f, [y, x], i) $ i = 0..2

                            3          2
                         3 x  + 2, 0, x
```

Alternatively:

```
>> coeff(f, y, i) $ i = 0..2

                            3          2
                         3 x  + 2, 0, x

>> delete f:
```

Example 4. In the same way, `coeff` may be applied to polynomials of type `DOM_POLY`:

```
>> p := poly(3*x^3 + x, [x], Dom::IntegerMod(7)):
   coeff(p)

                         3 mod 7, 1 mod 7

>> coeff(p, i) $ i = 0..3

               0 mod 7, 1 mod 7, 0 mod 7, 3 mod 7
```

For multivariate polynomials, the coefficients with respect to an indeterminate are polynomials in the other indeterminates:

```
>> p := poly(3*x^3 + x^2*y^2 + 2, [x, y]):

>> coeff(p, y, 0), coeff(p, y, 1), coeff(p, y, 2);

                 3                                2
        poly(3 x  + 2, [x]), poly(0, [x]), poly(x , [x])

>> coeff(p, x, 0), coeff(p, x, 1), coeff(p, x, 2)

                                                 2
            poly(2, [y]), poly(0, [y]), poly(y , [y])
```

Note that the indeterminates passed to `coeff` will be used, even if the polynomial provides different indeterminates :

```
>> coeff(p, z, 0), coeff(p, z, 1), coeff(p, z, 2)

         3    2  2
 poly(3 x  + x  y  + 2, [x, y]), poly(0, [x, y]),

    poly(0, [x, y])

>> delete p:
```

Example 5. The result of `coeff` is not fully evaluated:

```
>> p := poly(27*x^2 + a*x, [x]): a := 5:
   coeff(p, x, 1), eval(coeff(p, x, 1))

                               a, 5

>> delete p, a:
```

`coerce` – type conversion

`coerce(object, T)` tries to convert `object` into an element of the domain `T`.

Call(s):

- `coerce(object, T)`

Parameters:

`object`	—	any object
`T`	—	any domain

Return Value: an object of the domain `T`, or the value `FAIL`.

Overloadable by: `T`

Related Functions: `domtype`, `expr`, `testtype`, `type`

Details:

- `coerce(object, T)` tries to convert `object` to an element of the domain `T`. If this is not possible or not implemented, then `FAIL` is returned.
- Domains usually implement the two methods `"convert"` and `"convert_to"` for conversion tasks.

  `coerce` uses these methods in the following way: It first calls `T::convert(object)` to perform the conversion. If this call yields `FAIL`, then the result of the call `object::dom::convert_to(object, T)` is returned, which again may be the value `FAIL`.
- To find out the possible conversions for the `object` or which conversions are provided by the domain `T`, please read the description of the method `"coerce"` or `"convert"`, respectively, that can be found on the help page of the domain `T`, and the description of the method `"convert_to"` on the help page of the domain of `object`.
- Only few basic domains currently implement the methods `"convert"` and `"convert_to"`, but this will be extended in future versions of MuPAD.
- Use the function `expr` to convert an object into an element of a basic domain.
- Note that often a conversion can also be achieved by a call of the constructor of the domain `T`. See example 3.

Example 1. We start with the conversion of an array into a list of domain type `DOM_LIST`:

```
>> a := array(1..2, 1..3, [[1, 2, 3], [4, 5, 6]])
```

```
+-         -+
|  1, 2, 3  |
|           |
|  4, 5, 6  |
+-         -+
```

```
>> coerce(a, DOM_LIST)
```

```
[1, 2, 3, 4, 5, 6]
```

The conversion of an array into a polynomial is not implemented, and thus `coerce` returns `FAIL`:

```
>> coerce(a, DOM_POLY)
```

```
FAIL
```

One can convert a one- or two-dimensional array into a matrix, and vice versa. An example:

```
>> A := coerce(a, matrix); domtype(A)
```

```
+-         -+
|  1, 2, 3  |
|           |
|  4, 5, 6  |
+-         -+

Dom::Matrix()
```

The conversion of a matrix into a list is also possible. The result is then a list of inner lists, where the inner lists represent the rows of the matrix:

```
>> coerce(A, DOM_LIST)
```

```
[[1, 2, 3], [4, 5, 6]]
```

One can convert lists into sets, and vice versa. An example:

```
>> coerce([1, 2, 3, 2], DOM_SET)
```

```
{1, 2, 3}
```

Any MuPAD object can be converted into a string, such as the arithmetical expression `2*x + sin(x^2)`:

```
>> coerce(2*x + sin(x^2), DOM_STRING)
```

```
"2*x + sin(x^2)"
```

Example 2. The function `factor` computes a factorization of a polynomial expression and returns an object of the library domain `Factored`:

```
>> f := factor(x^2 + 2*x + 1);
   domtype(f)
```

```
                                   2
                            (x + 1)

                            Factored
```

This domain implements the conversion routine `"convert_to"`, which we can call directly to convert the factorization into a list (see `factor` for details):

```
>> Factored::convert_to(f, DOM_LIST)
```

```
                          [1, x + 1, 2]
```

However, it is more convenient to use `coerce`, which internally calls the slot routine `Factored::convert_to`:

```
>> coerce(f, DOM_LIST)
```

```
                          [1, x + 1, 2]
```

Example 3. Note that often a conversion can also be achieved by a call of the constructor of a domain `T`. For example, the following call converts an array into a matrix of the domain type `Dom::Matrix(Dom::Rational)`:

```
>> a := array(1..2, 1..2, [[1, 2], [3, 4]]):
   MatQ := Dom::Matrix(Dom::Rational):

>> MatQ(a)
```

```
                           +-      -+
                           |  1, 2  |
                           |        |
                           |  3, 4  |
                           +-      -+
```

The call `MatQ(a)` implies the call of the method `"new"` of the domain `MatQ`, which in fact calls the method `"convert"` of the domain `MatQ` to convert the array into a matrix.

Here, the same can be achieved with the use of `coerce`:

```
>> A := coerce(a, MatQ);
   domtype(A)
```

```
+-      -+
|  1, 2  |
|        |
|  3, 4  |
+-      -+
```

`Dom::Matrix(Dom::Rational)`

Note that the constructor of a domain `T` is supposed to *create* objects, not to convert objects of other domains into the domain type `T`. The constructor often allows more than one argument which allows to implement various user-friendly ways to create the objects (e.g., see the several possibilities for creating matrices offered by `matrix`).

`collect` – collect coefficients of a polynomial expression

`collect(p, x)` rewrites the polynomial expression `p` as $\sum_{i=0}^{n} a_i x^i$, such that x is not a polynomial indeterminate of any coefficient a_i.

`collect(p, [x1, x2, ...])` rewrites the polynomial expression `p` as

$$\sum_{i_1, i_2, \ldots} a_{i_1, i_2, \ldots} x_1^{i_1} x_2^{i_2} \cdots ,$$

such that none of the x_i is a polynomial indeterminate of any coefficient $a_{i_1, i_2, \ldots}$.

If a third argument `f` is given, then each coefficient in the return values above is replaced by $f(a_i)$ or $f(a_{i_1, i_2, \ldots})$, respectively.

Call(s):

- `collect(p, x <, f>)`
- `collect(p, [x1, x2, ...] <, f>)`

Parameters:

`p`	—	a polynomial expression
`x, x1, x2, ...`	—	the indeterminates of the polynomial: typically, identifiers or indexed identifiers
`f`	—	a function

Return Value: a polynomial expression, or `FAIL` if `p` cannot be converted into a polynomial.

Further Documentation: Chapter "Manipulating Expressions" of the Tutorial.

Related Functions: `coeff`, `combine`, `expand`, `factor`, `indets`, `normal`, `poly`, `rectform`, `rewrite`, `simplify`

Details:

- `collect` groups the terms in `p` with like powers of the given indeterminates together. `collect` returns a modified copy of `p`; the argument itself remains unchanged. See example 1.

- `collect` is merely a shortcut for the functional composition of `expr` and `poly`. It first uses `poly` to convert `p` into a polynomial in the given unknowns. This has the effect that the terms are collected. Then the result is again converted into a polynomial expression via `expr`. See the help page of `poly` for more information and examples.

- The indeterminates need not be identifiers or indexed identifiers. Any expression can be used as an indeterminate as long as it is neither rational nor constant. E.g., the expressions `sin(x)`, `f(x)`, or `y^(1/3)` are accepted as indeterminates, but the constant expressions `sin(1)` and `f(1)` are not allowed. More precisely, `x` is accepted as polynomial indeterminate if and only if the call `indets(x, PolyExpr)` returns `{x}` . See the help page of `indets` for more information, and also example 2.

- `collect` does not recursively collect the operands of non-polynomial subexpressions of `p`. See example 2.

- The terms in the result of `collect` are usually not ordered; use `poly` instead to achieve this.

  Note also that the "constant" terms corresponding to a_0 or $a_{0,0,...}$ are not always grouped together.

  See example 4.

- `collect` returns `FAIL` if `p` cannot be converted into a polynomial; the help of `poly` has more information when this is the case. See example 3.

Example 1. We define a polynomial expression `p` and collect terms with like powers of `x` and `y`:

```
>> p := x*y + z*x*y + y*x^2 - z*y*x^2 + x + z*x;
   collect(p, [x, y])

                                   2       2
            x + x y + x z + x y z + x  y - x  y z

                                   2
           x (z + 1) + x y (z + 1) + x  y (1 - z)
```

The expression `p` itself remains unchanged:

```
>> p
```

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

Now we collect terms with like powers of x:

```
>> collect(p, [x])
```

```
                         2
x (y + z + y z + 1) + x  (y - y z)
```

If there is only one indeterminate, then the square brackets may be omitted:

```
>> collect(p, x)
```

```
                         2
x (y + z + y z + 1) + x  (y - y z)
```

By passing the third argument factor, we cause every coefficient to be factored:

```
>> collect(p, x, factor)
```

```
                        2
x (y + 1) (z + 1) + x  y (1 - z)
```

Example 2. collect has the same behavior as poly with respect to non-polynomial subexpressions. Such a subexpression remains unchanged, even if it contains one of the given indeterminates. In particular, collect is not applied recursively to the operands of a non-polynomial subexpression:

```
>> collect(sin((x + 1)^2)*(x + 1) + 5*sin((x + 1)^2) + x, x)
```

```
              2                  2
6 sin((x + 1) ) + x (sin((x + 1) ) + 1)
```

However, a non-polynomial subexpression may be passed to collect as indeterminate, provided that it is accepted as indeterminate by poly:

```
>> collect(sin((x + 1)^2)*(x + 1) + 5*sin((x + 1)^2) + x,
           sin((x + 1)^2))
```

```
                        2
x + (x + 6) sin((x + 1) )
```

An error occurs if one of the indeterminates is illegal:

```
>> collect(1 + I*(x + I), I)
```

```
Error: Illegal indeterminate [poly];
during evaluation of 'collect'
```

In this example, you can use `rectform` to achieve the desired result:

```
>> rectform(1 + I*(x + I))

                     - Im(x) + I Re(x)
```

Example 3. `collect` returns `FAIL` if the input cannot be converted into a polynomial:

```
>> collect(1/x, x)

                           FAIL
```

Example 4. The terms in the result of `collect` are usually not ordered by increasing or decreasing degree:

```
>> collect(1 + x^2 + x, [x])

                              2
                         x + x  + 1
```

Use `poly` to achieve this:

```
>> poly(1 + x^2 + x, [x])

                         2
                    poly(x  + x + 1, [x])
```

Also, constant terms are not necessarily grouped together:

```
>> collect(sin(1) + (x + 1)^2, [x])

                                  2
                    2 x + sin(1) + x  + 1

>> poly(sin(y) + (x + 1)^2, [x])

                 2
          poly(x  + 2 x + (sin(y) + 1), [x])
```

`combine` – combine terms of the same algebraic structure

`combine(f)` tries to rewrite products of powers in the expression `f` as a single power.

`combine(f, target)` combines several calls to the target function(s) in the expression `f` to a single call.

Call(s):

- `combine(f)`
- `combine(f, target)`
- `combine(f, [target1, target2, ...])`

Parameters:

`f`	—	an arithmetical expression, an array, a list, a polynomial, or a set
`target`	—	one of the identifiers `arctan`, `exp`, `ln`, *`sincos`*, or *`sinhcosh`*

Return Value: an object of the same type as the input object `f`.

Side Effects: `combine` reacts to properties of identifiers appearing in the input.

Overloadable by: `f`

Further Documentation: Chapter "Manipulating Expressions" of the Tutorial.

Related Functions: `denom`, `expand`, `factor`, `normal`, `numer`, `radsimp`, `rectform`, `rewrite`, `simplify`

Details:

- `combine(f)` applies the following rewriting rules to products of powers occurring as subexpressions in an arithmetical expression `f`:

    - $x^a x^b = x^{a+b}$
    - $x^b y^b = (xy)^b$
    - $(x^a)^b = x^{ab}$

The last two rules are only valid under certain additional restrictions, e.g., when b is an integer. Except for the third rule, this behavior of `combine` is the inverse functionality of `expand`. See example 1.

Since MuPAD's internal simplifier automatically applies the above rules in the reverse direction in certain cases, `combine` sometimes has no effect. See example 2.

NOTE

`combine(f, target)` applies rewriting rules applicable to the target function(s) to an arithmetical expression `f`. Some of the rules are only valid under certain additional restrictions. With respect to most of the rules, `combine` implements the inverse functionality of `expand`. Here is a list of the rewriting rules for the various targets:

`target = arctan`:

- $\arctan(x) + \arctan(y) = \arctan\left(\frac{x+y}{1-xy}\right)$

`target = exp` (see example 4:)

- $\exp(a)\exp(b) = \exp(a+b)$
- $\exp(a)^b = \exp(ab)$ (where valid, reacting to properties)

`target = ln` (see example 5:)

- $\ln(a) + \ln(b) = \ln(ab)$
- $b\ln(a) = \ln(a^b)$

`target =` *`sincos`* (see example 3):

- $\sin(x)\sin(y) = \frac{1}{2}\cos(x-y) - \frac{1}{2}\cos(x+y)$
- similar rules for $\sin(x)\cos(y)$ and $\cos(x)\cos(y)$
- the rules above are applied recursively to powers of sin and cos with positive integral exponents

`target =` *`sinhcosh`*:

- $\sinh(x)\sinh(y) = \frac{1}{2}\cosh(x+y) - \frac{1}{2}\cosh(x-y)$
- similar rules for $\sinh(x)\cosh(y)$ and $\cosh(x)\cosh(y)$
- the rules above are applied recursively to powers of sinh and cosh with positive integral exponents

- `combine` works recursively on the subexpressions of `f`.
- If the second argument is a list of targets, then `combine` is applied to `f` subsequently for each of the targets in the list. See example 6.
- If `f` is array, a list, or a set, `combine` is applied to all entries of `f`; see example 7. If `f` is a polynomial or a series expansion, of type `Series::Puiseux` or `Series::gseries`, `combine` is applied to each coefficient; see example 8.

Example 1. Without a second argument, `combine` combines powers of the same base:

```
>> combine(sin(x) + x*y*x^(exp(1)))

                        exp(1) + 1
          sin(x) + y x
```

Moreover, `combine` also combines powers with the same exponent in certain cases:

```
>> combine(sqrt(2)*sqrt(3))

                        1/2
                       6
```

Example 2. In most cases, however, `combine` does not combine powers with the same exponent:

```
>> combine(y^5*x^5)

                         5  5
                        x  y
```

Example 3. With the second argument *sincos*, `combine` rewrites products of sines and cosines as a sum of sines and cosines with more complicated arguments:

```
>> combine(sin(a)*cos(b) + sin(b)^2, sincos)

       sin(a + b)   cos(2 b)   sin(a - b)
       ---------- - -------- + ---------- + 1/2
           2           2           2
```

Note that powers of sines or cosines with negative integer exponents are not rewritten:

```
>> combine(sin(b)^(-2), sincos)

                              1
                           -------
                                2
                           sin(b)
```

Example 4. With the second argument `exp`, the well-known rules for the exponential function are applied:

```
>> combine(exp(3)*exp(2), exp)

                           exp(5)

>> combine(exp(a)^2, exp)

                          exp(2 a)
```

Example 5. This example shows the application of rules for the logarithm, and at the same time the dependence on properties of the identifiers appearing in the input. The logarithm of a product does not always equal the sum of the logarithms of its factors; but for positive numbers, this rule may be applied:

```
>> combine(ln(a)+ln(b), ln)

                        ln(a) + ln(b)

>> assume(a>0): assume(b>0):
   combine(ln(a)+ln(b), ln)

                           ln(a b)

>> unassume(a): unassume(b):
```

Example 6. The second argument may also be a list of targets. Then the rewriting rules for each of the targets in the list are applied:

```
>> combine(ln(2)+ln(3)+sin(a)*cos(a), [ln, sincos])

                              sin(2 a)
                     ln(6) + --------
                                 2
```

Example 7. `combine` maps to sets:

```
>> combine({sqrt(2)*sqrt(5), sqrt(2)*sqrt(11)})

                           1/2    1/2
                        {10   , 22   }
```

Example 8. `combine` maps to the coefficients of polynomials:

```
>> combine(poly(sin(x)*cos(x)*y, [y]), sincos)

                         / / sin(2 x) \        \
                     poly| | -------- | y, [y] |
                         \ \     2    /        /
```

However, it does not touch the polynomial's indeterminates:

```
>> combine(poly(sin(x)*cos(x)), sincos)

              poly(sin(x) cos(x), [sin(x), cos(x)])
```

Background:

- Advanced users can extend the functionality of `combine` by implementing additional rewriting rules for other target functions. This works by defining a new slot `"target"` of `combine`; you need to unprotect the identifier `combine` first in order to do that. Afterwards, the command `combine(f, target)` leads to the call `combine::target(f)` of the corresponding slot routine.
- By default, `combine` handles a subexpression `g(x1,x2,...)` of `f` by calling itself recursively for the operands `x1`, `x2`, etc. Users can change this behavior for their own mathematical function given by a function environment `g` by implementing a `"combine"` slot of `g`. To handle the subexpression `g(x1,x2,...)`, `combine` then calls the slot routine `g::combine` with the argument sequence `x1,x2,...` of `g`.

Changes:

- The code for the target `ln` was rewritten and enhanced.

`copyClosure` – copies the lexical closure of a procedure

`copyClosure(f)` copies the lexical closure of a procedure or procedure environment `f`.

Call(s):

- `copyClosure(f)`

Parameters:

`f` — a procedure or procedure environment to be copied

Return Value: the copied procedure or procedure environment

Related Functions: `_assign`

Details:

- Usually, when a procedure is copied, for example by assigning it to an identifier, the lexical closure of the procedure is not copied. Via the copied procedure one can change the lexical closure of the original procedure. Thus, the lexical closure of a procedure shows the so-called *reference effect*.
- `copyClosure` may be used to copy the lexical closure of a procedure. Changes in the closure of the copy no longer affect the original procedure's closure.
- Closures are implemented by procedure environments (kernel type `DOM_PROC_ENV`) in MuPAD. `copyClosure` works by copying all lexically enclosing procedure environments of a procedure.
- `copyClosure` may also be used to copy a procedure environment and all its lexically enclosing environments only.

Example 1. Procedure closures show the reference effect: The procedure `f` generated by `gen` changes its closure via the variable `i`. A "normal" copy `g` of `f` changes the variable in the same closure, as is seen by repeatedly calling `f` versus `g`.

```
>> gen:= proc()
       option escape;
       local i;
   begin
       i := 0;
       proc() begin i := i+1 end
   end:
>> f := gen():
   g := f:
   f(), g(), f(), g()
```

```
                        1, 2, 3, 4
```

If one now generates `f` again by calling `gen`, but copies `g` by calling `copyClosure`, then `g` has its own closure and now longer changes the variable `i` in the closure of `f`.

```
>> f := gen():
   g := copyClosure(f):
   f(), g(), f(), g()

                         1, 1, 2, 2
```

Changes:

- `copyClosure` is a new function.

`complexInfinity` – complex infinity

`complexInfinity` represents the only non-complex point of the one-point compactification of the complex numbers.

Related Functions: `infinity`

Details:

- Mathematically, `complexInfinity` is the north pole of the Riemann sphere, with the unit circle as equator and the point 0 at the south pole.
- With respect to arithmetic, `complexInfinity` behaves like "`1/0`". In particular, nonzero complex numbers may be multiplied or divided by `complexInfinity` or `1/complexInfinity`. Adding `complexInfinity` to a finite number yields again `complexInfinity`.
- With respect to arithmetical operations, `complexInfinity` is incompatible with the real `infinity`.

Example 1. `complexInfinity` can be used in arithmetical operations with complex numbers. The result in multiplications or divisions is either `complexInfinity`, `0`, or `undefined`:

```
>> 3*complexInfinity, I*complexInfinity, 0*complexInfinity;
   3/complexInfinity, I/complexInfinity, 0/complexInfinity;
   complexInfinity/3, complexInfinity/I;
   complexInfinity*complexInfinity, complexInfinity/complexInfinity;
```

```
complexInfinity, complexInfinity, undefined

0, 0, 0

complexInfinity, complexInfinity

complexInfinity, undefined
```

The result in additions is `undefined` if one of the operands is infinite, and `complexInfinity` otherwise:

```
>> complexInfinity + complexInfinity, infinity + complexInfinity;
   3 + complexInfinity, I + complexInfinity, PI + complexInfinity

undefined, undefined

complexInfinity, complexInfinity, complexInfinity
```

Symbolic expressions in arithmetical operations involving `complexInfinity` are implicitly assumed to be different from both `0` and `complexInfinity`:

```
>> delete x:
   x*complexInfinity, x/complexInfinity, complexInfinity/x,
   x + complexInfinity

complexInfinity, 0, complexInfinity, complexInfinity
```

Background:

- `complexInfinity` is the only element of the domain `stdlib::CInfinity`.

Changes:

- Addition of a finite number to `complexInfinity` yields again `complexInfinity`.

`conjugate` – complex conjugation

`conjugate(z)` computes the conjugate $\Re(z) - i\,\Im(z)$ of a complex number $z = \Re(z) + i\,\Im(z)$.

Call(s):

- `conjugate(z)`

Parameters:

`z` — an arithmetical expression

Return Value: an arithmetical expression.

Overloadable by: `z`

Side Effects: `conjugate` is sensitive to properties of identifiers set via `assume`.

Related Functions: `abs`, `assume`, `Im`, `Re`, `rectform`, `sign`

Details:

- For numbers of type `DOM_INT`, `DOM_RAT`, `DOM_FLOAT`, or `DOM_COMPLEX`, the conjugate is computed directly and very efficiently.
- `conjugate` can handle symbolic expressions. Properties of identifiers are taken into account (see `assume`). An identifier `z` without any property is assumed to be complex, and the symbolic call `conjugate(z)` is returned. See example 2.
- `conjugate` knows how to handle special mathematical functions, such as:

`_mult`	`_plus`	`_power`	`abs`	`cos`	`cosh`	`cot`
`coth`	`csc`	`csch`	`erf`	`erfc`	`exp`	`gamma`
`igamma`	`sec`	`sech`	`sin`	`sinh`	`tan`	`tanh`

  See example 1.
- If `conjugate` does not know how to handle a special mathematical function, then a symbolic `conjugate` call is returned. See example 3.

Example 1. `conjugate` knows how to handle sums, products, the exponential function and the sine function:

```
>> conjugate((1 + I)*exp(2 - 3*I))

                    (1 - I) exp(2 + 3 I)

>> delete z: conjugate(z + 2*sin(3 - 5*I))

                 conjugate(z) + 2 sin(3 + 5 I)
```

Example 2. `conjugate` reacts to properties of identifiers:

```
>> delete x, y: assume(x, Type::Real):
   conjugate(x), conjugate(y)

                       x, conjugate(y)
```

Example 3. If the input contains a function that the system does not know, then a symbolic `conjugate` call is returned:

```
>> delete f, z: conjugate(f(z) + I)

                        conjugate(f(z)) - I
```

Now suppose that `f` is some user-defined mathematical function, and that $\overline{f(z)} = f(\overline{z})$ holds for all complex numbers z. To extend the functionality of `conjugate` to `f`, we embed it into a function environment and suitably define its `"conjugate"` slot:

```
>> f := funcenv(f):
   f::conjugate := u -> f(conjugate(u)):
```

Now, whenever `conjugate` is called with an argument of the form `f(u)`, it calls `f::conjugate(u)`, which in turn returns `f(conjugate(u))`:

```
>> conjugate(f(z) + I), conjugate(f(I))

                  f(conjugate(z)) - I, f(-I)
```

Background:

- If a subexpression of the form `f(u,..)` occurs in `z` and `f` is a function environment, then `conjugate` attempts to call the slot `"conjugate"` of `f` to determine the conjugate of `f(u,..)`. In this way, you can extend the functionality of `conjugate` to your own special mathematical functions.

  The slot `"conjugate"` is called with the arguments `u,..` of `f`.

  If `f` has no slot `"conjugate"`, then the subexpression `f(u,..)` is replaced by the symbolic call `conjugate(f(u...))` in the returned expression.

  See example 3.

- Similarly, if an element `d` of a library domain `T` occurs as a subexpression of `z`, then `conjugate` attempts to call the slot `"conjugate"` of that domain with `d` as argument to compute the conjugate of `d`.

  If `T` does not have a slot `"conjugate"`, then `d` is replaced by the symbolic call `conjugate(d)` in the returned expression.

  The same happens for objects of kernel domains that are not arithmetical expressions, such as lists, arrays, tables, sets, or polynomials.

`contains` – test if an entry exists in a container

`contains(s, object)` tests if `object` is an element of the set `s`.

`contains(l, object)` returns the index of `object` in the list `l`.

`contains(t, object)` tests if the array, table, or domain `t` has an entry corresponding to the index `object`.

Call(s):

- `contains(s, object)`
- `contains(l, object <, i>)`
- `contains(t, object)`

Parameters:

`s`	—	a set
`l`	—	a list
`t`	—	an array, a table, or a domain
`object`	—	an arbitrary MuPAD object
`i`	—	an integer

Return Value: For sets, arrays, tables, or domains, `contains` returns one of the Boolean values `TRUE` or `FALSE`. For lists, the return value is a nonnegative integer.

Overloadable by: `s, l, t`

Related Functions: `_in`, `_index`, `has`, `op`, `slot`

Details:

- `contains` is a fast membership test for MuPAD's basic container data types. For lists and sets, `contains` searches the elements for the given object. However, for arrays, tables, and domains, `contains` searches the indices.
- `contains` works syntactically, i.e., mathematically equivalent objects are considered to be equal only if they are syntactically identical. `contains` does *not* represent elementhood in the mathematical sense. See example 2.
- `contains` does not descend recursively into subexpressions; use `has` to achieve this. See example 3.
- `contains(s, object)` returns `TRUE` if `object` is an element of the set `s`. Otherwise, it returns `FALSE`.
- `contains(l, object)` returns the position of `object` in the list `l` as a positive integer if `object` is an entry of `l`. Otherwise, the return value is `0`. If more than one entry of `l` is equal to `object`, then the index of the first occurrence is returned.

By passing a third argument `i` to `contains`, you can specify a position in the list where the search is to start. Then entries with index less than `i` are not taken into account. If `i` is out of range, then the return value is `0`.

Cf. examples 4 and 5.

- `contains(t, object)` returns `TRUE` if the array, table, or domain `t` has an entry corresponding to the index `object`. Otherwise, it returns `FALSE`. Cf. example 6.

- `contains` is a function of the system kernel.

Example 1. `contains` may be used to test if a set contains a given element:

```
>> contains({a, b, c}, a), contains({a, b, c}, 2)

                    TRUE, FALSE
```

Example 2. `contains` works syntactically, i.e., mathematically equivalent objects are considered to be equal only if they are syntactically identical. In this example `contains` returns `FALSE` since `y*(x + 1)` and `y*x + y` are different representations of the same mathematical expression:

```
>> contains({y*(x + 1)}, y*x + y)

                    FALSE
```

Elementhood in the mathematical sense is represented by the operator `in`:

```
>> property::simpex(y*x + y in {y*(x+1)})

                    TRUE
```

Example 3. `contains` does not descend recursively into the operands of its first argument. In the following example, `c` is not an element of the set, and therefore `FALSE` is returned:

```
>> contains({a, b, c + d}, c)

                    FALSE
```

If you want to test whether a given expression is contained *somewhere inside* a complex expression, please use `has`:

```
>> has({a, b, c + d}, c)

                    TRUE
```

Example 4. `contains` applied to a `list` returns the position of the specified object in the list:

```
>> contains([a, b, c], b)
```

2

If the list does not contain the object, `0` is returned:

```
>> contains([a, b, c], d)
```

0

Example 5. `contains` returns the position of the first occurrence of the given object in the list if it occurs more than once:

```
>> l := [a, b, a, b]: contains(l, b)
```

2

A starting position for the search may be given as optional third argument:

```
>> contains(l, b, 1), contains(l, b, 2),
   contains(l, b, 3), contains(l, b, 4)
```

2, 2, 4, 4

If the third argument is out of range, then the return value is `0`:

```
>> contains(l, b, -1), contains(l, b, 0), contains(l, b, 5)
```

0, 0, 0

Example 6. For tables, `contains` returns `TRUE` if the second argument is a valid index in the table. The entries stored in the table are not considered:

```
>> t := table(13 = value): contains(t, 13), contains(t, value)
```

TRUE, FALSE

Similarly, `contains` tests if an `array` has a value for a given index. The array `a` has a value corresponding to the index `(1, 1)`, but none for the index `(1, 2)`:

```
>> a := array(1..3, 1..2, (1, 1) = x, (2, 1) = PI):
   contains(a, (1, 1)), contains(a, (1, 2))
```

TRUE, FALSE

`contains` is not intended for testing if an array contains a given value:

```
>> contains(a, PI)
```

```
 Error: Index dimension mismatch [array]
```

Even if the dimensions match, the index must not be out of range:

```
>> contains(a, (4, 4))
```

```
 Error: Illegal argument [array]
```

Example 7. `contains` may be used to test, whether a `domain` has the specified `slot`:

```
>> T := newDomain("T"):  T::index := value:
   contains(T, index), contains(T, value)
```

```
                              FALSE, FALSE
```

There is no entry corresponding to the slot `index` in `T`. Please keep in mind that the syntax `T::index` is equivalent to `slot(T, "index")`:

```
>> contains(T, "index")
```

```
                                  TRUE
```

Example 8. Users can overload `contains` for their own domains. For illustration, we create a new domain `T` and supply it with an `"contains"` slot, which tests is the set of entries of an element contains the given value `idx`:

```
>> T := newDomain("T"):
   T::contains := (e, idx) -> contains({extop(e)}, idx):
```

If we now call `contains` with an object of domain type `T`, the slot routine `T::contains` is invoked:

```
>> e := new(T, 1, 2): contains(e, 2), contains(e, 3)
```

```
                               TRUE, FALSE
```

`content` – the content of a polynomial

`content(p)` computes the content of the polynomial `p`, i.e., the gcd of its coefficients.

Call(s):

- `content(p)`
- `content(f <, vars>)`

Parameters:

`p`	—	a polynomial of type `DOM_POLY`
`f`	—	a polynomial expression
`vars`	—	a list of indeterminates of the polynomial: typically, identifiers or indexed identifiers

Return Value: an arithmetical expression, or the value `FAIL`.

Overloadable by: `p`

Related Functions: `coeff`, `factor`, `gcd`, `icontent`, `ifactor`, `igcd`, `ilcm`, `lcm`, `poly`, `polylib::primpart`

Details:

- If `p` is the zero polynomial, then `content` returns 0.
- If `p` is a nonzero polynomial with coefficient ring `IntMod(n)` and `n` is a prime number, then `content` returns 1. If `n` is not a prime number, an error message is issued.
- If `p` is a polynomial with a library domain `R` as coefficient ring, the `gcd` of its coefficients is computed using the slot `gcd` of `R`. If no such slot exists, then `content` returns `FAIL`.
- If `p` is a polynomial with coefficient ring `Expr`, then `content` does the following.

  If all coefficients of `p` are either integers or rational numbers, `content(p)` is equivalent to `gcd(coeff(p))`, and the return value is a positive integer or rational number. See example 1.

  If at least one coefficient is a floating point number or a complex number and all other coefficients are numbers, then `content` returns 1. See example 2.

  If at least one coefficient is not a number and all coefficients of `p` can be converted into polynomials via `poly`, then `content(p)` is equivalent to `gcd(coeff(p))`. See example 3.

  Otherwise, `content` returns 1.
- A polynomial expression `f` is converted into a polynomial with coefficient ring `Expr` via `p := poly(f <, vars>)`, and then `content` is applied to `p`. See example 1.

- Use `icontent` for polynomials that are known to have integer or rational coefficients, since it is much faster than `content`.
- Dividing the coefficients of `p` by its content gives its primitive part. This one can also be obtained directly using `polylib::primpart`.

Example 1. If `p` is a polynomial with integer or rational coefficients, the result is the same as for `icontent`:

```
>> content(poly(6*x^3*y + 3*x*y + 9*y, [x, y]))

                                3
```

The following call, where the first argument is a polynomial expression and not a polynomial, is equivalent to the one above:

```
>> content(6*x^3*y + 3*x*y + 9*y, [x, y])

                                3
```

If no list of indeterminates is specified, then `poly` converts the expression into a polynomial with respect to all occurring indeterminates, and we obtain yet another equivalent call:

```
>> content(6*x^3*y + 3*x*y + 9*y)

                                3
```

Above, we considered the polynomial as a bivariate polynomial with integer coefficients. We can also consider the same polynomial as a univariate polynomial in `x`, whose coefficients contain a parameter `y`. Then the coefficients and their gcd—the content—are polynomial expressions in `y`:

```
>> content(poly(6*x^3*y + 3*x*y + 9*y, [x]))

                               3 y
```

Here is another example where the coefficients and the content are again polynomial expressions:

```
>> content(poly(4*x*y + 6*x^3 + 6*x*y^2 + 9*x^3*y, [x]))

                             3 y + 2
```

The following call is equivalent to the previous one:

```
>> content(4*x*y + 6*x^3 + 6*x*y^2 + 9*x^3*y, [x])

                             3 y + 2
```

Example 2. If a polynomial or polynomial expression has numeric coefficients and at least one floating point number is among them, its content is 1:

```
>> content(2.0*x+2.0)

                                   1
```

Example 3. If not all of the coefficients are numbers, the gcd of the coefficients is returned:

```
>> content(poly(x^2*y+x, [y]))

                                   x
```

`context` – evaluate an object in the enclosing context

Within a procedure, `context(object)` evaluates `object` in the context of the calling procedure.

Call(s):

- `context(object)`

Parameters:

`object` — any MuPAD object

Return Value: the evaluated object.

Side Effects: `context` is sensitive to the value of the environment variable `LEVEL`, which determines the maximal substitution depth for identifiers.

Related Functions: `DOM_PROC`, `eval`, `freeze`, `hold`, `LEVEL`, `level`, `MAXLEVEL`, `proc`

Details:

- Most MuPAD procedures evaluate their arguments before executing the body of the procedure. However, if the procedure is declared with option `hold`, then the arguments are passed to the procedure unevaluated. `context` serves to evaluate such arguments *a posteriori* from within the procedure.

- Like most MuPAD procedures, `context` first evaluates its argument `object` as usual in the context of the current procedure. Then the result is evaluated again in the dynamical context that was valid before the current procedure was called. The enclosing context is either the interactive level or the procedure that called the current procedure.

- `"func_call"`-methods of domains never evaluate their arguments, whether the option `hold` is used or not. See example 2.

- `context` is sensitive to the value of the environment variable `LEVEL`, which determines the maximal depth of the recursive process that replaces an identifier by its value during evaluation. The evaluation of the argument takes place with the value of `LEVEL` that is valid in the current procedure, which is `1` by default. The second evaluation uses the value of `LEVEL` that is valid in the enclosing context, which is usually `1` if the enclosing context is also a procedure, while it is `100` by default if the enclosing context is the interactive level. See example 3.

- The function `context` must not be called at interactive level, and `context` calls must not be nested. Thus it is not possible to evaluate an object in higher levels of the dynamical call stack. See example 4.

- `context` is a function of the system kernel.

Example 1. We define a procedure `f` with option `hold`. If this procedure is called with an identifier as argument, such as `a` below, the `identifier` itself is the actual argument inside of `f`. `context` may be used to get the value of `a` in the outer context:

```
>> a := 2:
   f := proc(i)
          option hold;
        begin
          print(i, context(i), i^2 + 2, context(i^2 + 2));
        end_proc:
   f(a):

                  2
         a, 2, a  + 2, 6
```

If a procedure with option `hold` is called from another procedure you will see strange effects if the procedure with option `hold` does not evaluate its formal parameters with `context`. Here, the value of the formal parameter `j` in `g` is the `variable i` which is defined in the context of procedure `f` and not its value `4`. When you want to access the value of this variable you have to use `context`, otherwise you see the output `DOM_VAR(0,2)` which is the variable `i` of `f` which has lost its scope:

```
>> f := proc()
         local i;
       begin
         i := 4:
         g(i);
       end_proc:
   g := proc(j)
         option hold;
       begin
         print(j, eval(j), context(j));
         print(j + 1)
       end_proc:
   f()
```

```
                DOM_VAR(0,2), DOM_VAR(0,2), 4

                      DOM_VAR(0,2) + 1
```

Example 2. The "func_call" method of a domain is implicitly declared with option hold. We define a "func_call" method for the domain DOM_STRING of MuPAD strings. The slot routine converts its remaining arguments into strings and appends them to the first argument, which coincides with the string that is the 0th operand of the function call:

```
>> unprotect(DOM_STRING):
   DOM_STRING::func_call :=
     string -> _concat(string, map(args(2..args(0)), expr2text)):
   a := 1:  "abc"(1, a, x)
```

```
                          "abc1ax"
```

You see that the identifier a was added to the string, and not its value 1. Use context to access the value that a has before the "func_call" method is invoked:

```
>> DOM_STRING::func_call :=
     string -> _concat(string, map(context(args(2..args(0))),
                                   expr2text)):
   "abc"(1, a, x);
   delete DOM_STRING::func_call:  protect(DOM_STRING, Error):
```

```
                          "abc11x"
```

Example 3. This example shows the influence of the environment variable `LEVEL` on the evaluation of `context` and the differences to the functions `eval` and `level`. `p` is a function with option `hold`. `x` is a formal parameter of this procedure. When evaluating their arguments `context`, `eval` and `level` all replace `x` first by its value `a`. Then `eval` evaluates `a` in the current context with `LEVEL = 1` and yields the value `b`. `context` evaluates `a` in the enclosing context (which is the interactive level) with `LEVEL = 100` and yields `c`. `level` always returns the result of the first evaluation step, which is `a`.

When the `LEVEL` of the interactive level is `1`, `context` returns `b` like `eval` since the second evaluation is performed with `LEVEL = 1` like in `eval`.

The local variable `b` of `p` does not influence the evaluation in `context`, `eval` and `level` since it is only a locally declared variable of type `DOM_VAR` which has nothing to do with the identifier `b`, which is the value of `a`:

```
>> delete a, b, c:  a := b:  b := c:
   p := proc(x)
          option hold;
          local b;
        begin
          b := 2;
          eval(x), context(x), level(x), level(x,2);
        end:
   p(a);
   LEVEL := 1: p(a);
   delete LEVEL:

                            b, c, a, a

                            b, b, a, a
```

Example 4. The function `context` must not be called at interactive level:

```
>> context(x)

 Error: Function call not allowed on interactive level [context]
```

`contfrac` – the domain of continued fractions

`contfrac(r)` creates a continued fraction approximation of the real number `r`.

`contfrac(f, x = x0)` creates a continued fraction approximation of the expression `f` as a function of `x` around `x = x0`.

Creating Elements:

- `contfrac(r <, n>)`
- `contfrac(f, x <, m>)`
- `contfrac(f, x = x0 <, m>)`

Parameters:

`r`	—	a real number or a numerical expression that can be converted to a real floating point number
`n`	—	the number of significant decimal digits: a positive integer. The default value is `n` = `DIGITS`.
`f`	—	an arithmetical expression interpreted as a function of x
`x`	—	an identifier
`x0`	—	the expansion point: an arithmetical expression, ±`infinity` or `complexInfinity`. The default value is 0.
`m`	—	the 'number of terms': a positive integer. The default value is `m` = `ORDER`.

Related Functions: `numlib::contfrac`, `series`, `Series::Puiseux::contfrac`

Return Value: The call `contfrac(r <, n>)` with a numerical value `r` returns an object of type `numlib::contfrac`. The call `contfrac(f, x = x0 <, m>)` with a symbolic expression `f` returns an object of type `contfrac`. `FAIL` is returned if no series expansion of `f` around `x0` could be computed.

Side Effects: When called with an irrational numerical value `r`, the function is sensitive to the environment variable `DIGITS` which determines the numerical working precision. For symbolic expressions `f`, the function is sensitive to the environment variable `ORDER` which determines the number of terms in truncated series expansions.

Overloadable by: `r, f`

Details:

- The continued fraction expansion `contfrac(r <, n>)` of a real number or numerical expression `r` is an expansion of the form

$$a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{\ddots\, a_{k-1} + \cfrac{1}{a_k + \cdots}}}}$$

  where `a1` is the integer `floor(r)` and `a2`, `a3`, `...` are positive integers.

The continued fraction is computed by `numlib::contfrac(r <, n>)`; the expansion returned by `contfrac` is of domain type `numlib::contfrac`.

See the documentation of `numlib::contfrac` for further details.

- A continued fraction expansion `contfrac(f, x = x0)` of a symbolic expression `f` in the indeterminate `x` is an expansion of the form

$$a_1 + \cfrac{(x-x_0)^{e_1}}{a_2 + \cfrac{(x-x_0)^{e_2}}{a_3 + \cfrac{(x-x_0)^{e_3}}{\ddots\, a_{k-1} + \cfrac{(x-x_0)^{e_{k-1}}}{a_k + O((x-x_0)^{e_k})}}}}$$

  where

  - $a_1, \ldots, a_k$ are arithmetical expressions not containing powers of $x - x_0$. The coefficients $a_2, \ldots, a_k$ are nonzero.
  - e_1 is a rational number and $e_2, \ldots, e_k$ are positive rational numbers. If $a_1 \neq 0$, then e_1 is positive as well.

  If `x0` $= \pm\infty$ or `x0 = complexInfinity`, the terms $(x - x_0)^{e_i}$ have to be replaced by x^{-e_i}.

  For symbolic expressions `f`, `contfrac(f, x = x0)` returns an expansion of domain type `contfrac`.

- One may also call `contfrac(f)` without specifying an identifier `x`. In this case, `contfrac` extracts the indeterminates in `f` automatically via `indets`. `FAIL` is returned if more than one indeterminate is found.

  If `m` is not specified, the default value `m` = `ORDER` is used.

- `contfrac` uses the function `Series::Puiseux::contfrac` to compute the continued fraction in the symbolic case. If `f` is a rational function with respect to the expansion variable `x`, and the 'truncation order' `m` is not specified, then `contfrac` returns an exact continued fraction expansion of `f`. Cf. example 3.

Mathematical Methods

Method `rational`: rational representation

`rational(`*`contfrac`* `cf <,` *`positive integer`* `m>)`

- This method returns the rational function that is represented by the continued fraction `cf`. Only the coefficients 1 through `m` of `cf` are taken into account.

  If `m` is not specified, all coefficients of `cf` are taken into account. The return value of this method coincides with `op(cf, 2)`.

Method `_plus`: addition

`_plus(`*any* `cf,` *any* `cg)`

- At least one argument must be of type `contfrac`. All other arguments are converted via the constructor if possible. This method converts `cf` and `cg` into rational functions, adds them, and returns the sum converted back into a continued fraction. It returns `FAIL` if either the variables or the expansion points in the continued fractions `cf` and `cg` are different.

  If all terms in the sum are exact representations of rational functions, the continued fraction expansion of the sum is also an exact representation.

- This method overloads the function `_plus` of the system kernel.

Method `_mult`: multiplication

`_mult(`*any* `cf,` *any* `cg)`

- At least one argument must be of type `contfrac`. All other arguments are converted via the constructor if possible. This method converts `cf` and `cg` into rationals functions, multiplies them, and returns the product converted back into a continued fraction. It returns `FAIL` if either the variables or the expansion points in the continued fractions `cf` and `cg` are different.

  If all terms in the product are exact representations of rational functions, the continued fraction expansion of the product is also an exact representation.

- This method overloads the function `_mult` of the system kernel.

Method `_power`: powers of a continued fraction

`_power(`*contfrac* `cf,` *rational* `p)`

- This method converts `cf` into a rational function, computes the p-th power and returns the result as a continued fraction expansion.

  If `cf` is an exact representation of a rational function and `cf^p` is a rational function, the continued fraction expansion of the power is also an exact representation.

- This method overloads the function `_power` of the system kernel.

Method `series`: series of a continued fraction

series(*contfrac* cf <, *positive integer* m>)

series(*contfrac* cf, *indeterminate* x < = x0> <, *positive integer* m>)

- Computes the series expansion of the continued fraction `cf`. This is the series of the rational expression `op(cf, 2)` represented by `cf` around `x = x0`. It is returned as an element of the domain `Series::Puiseux`.
- If `x` is not specified, the default series variable is `op(cf, 3)`. If `x0` is not specified, the default expansion point is `op(cf, 4)`. If no 'number of terms' `m` is specified, `m` = `ORDER` is used.
- This method overloads the function `series`.

Access Methods

Method `nthcoeff`: the nth coefficient of the continued fraction

nthcoeff(*contfrac* cf, *positive integer* n)

- The call `nthcoeff(cf, n)` returns the n-th coefficient a_n of the continued fraction `cf` $= a_1 + \frac{(x-x_0)^{e_1}}{a_2 + \frac{(x-x_0)^{e_2}}{a_3 + \dots}}$. For $n > 1$, the coefficients cannot be zero.

  If `n` exceeds the actual number of coefficients in `cf`, the value `FAIL` is returned.
- This method overloads the function `nthcoeff` of the system kernel.

Method `nthterm`: the nth term of the continued fraction

nthterm(*contfrac* cf, *positive integer* n)

- The call `nthterm(cf, n)` returns the n-th term $(x - x_0)^{e_n}$ of the continued fraction `cf` $= a_1 + \frac{(x-x_0)^{e_1}}{a_2 + \frac{(x-x_0)^{e_2}}{a_3 + \dots}}$.

  If `n` exceeds the actual number of coefficients in `cf`, the value `FAIL` is returned.

Method `op`: the operands of the continued fraction

op(*contfrac* cf <, *nonnegative integer* n>)

- An object created by contfrac has the following operands. The 0-th operand op(cf, 0) is the domain type: either contfrac or numlib::contfrac. In the case of symbolic continued fractions of type contfrac, the first operand is the list of coefficient/term pairs $[a_i, (x - x_0)^{e_i}]$. The second operand is the rational function represented by these coefficients (this is the value returned by the method contfrac::rational). The third operand is the variable x of the expansion. The fourth operand is the expansion point x0.

Example 1. We compute some continued fraction expansions of real numbers:

```
>> contfrac(27/31), contfrac(PI, 5)

               1                        1
    -------------------, 3 + ----------------
               1                        1
    1 + ---------------      7 + ------------
                 1                        1
        6 + -----------         15 + -------
                   1                  1 + ...
            1 + -------
                3 + ...
```

They can also be computed by direct calls to numlib::contfrac:

```
>> numlib::contfrac(27/31), numlib::contfrac(PI, 5)

               1                        1
    -------------------, 3 + ----------------
               1                        1
    1 + ---------------      7 + ------------
                 1                        1
        6 + -----------         15 + -------
                   1                  1 + ...
            1 + -------
                3 + ...
```

Example 2. We compute symbolic continued fractions of functions:

```
>> contfrac(exp(x), x = 0), contfrac(exp(-3*x^2), x = 0)

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

```
                  x                - 1/3 + ---------------
- 2 + -----------------                                  2
                     x                                  x
      - 3 + -----------                   - 2 + ---------
                       x                                2
            2 + --------                        1 + O(x )
                5 + O(x)
```

If no expansion variable is specified, the symbolic expression to be expanded must be univariate:

```
>> contfrac(exp(x*y))
```

```
Error: 1st argument: the expression is not univariate [contfra\
c::function]
```

Symbolic parameters are accepted if the expansion variable is specified:

```
>> contfrac(exp(x*y), x)
```

```
                          x
1 + ------------------------------------
      -1                  x
     y   + -------------------------------
                          x
           - 2 + --------------------------
                      -1           x
                 - 3 y   + ----------------
                                  x
                           2 + ------------
                                    -1
                               5 y   + O(x)
```

In the next call, we specify the expansion point `x = 1` and request a specific 'number of terms' by the third argument:

```
>> contfrac(exp(x*y), x = 1, 3);
```

```
                        x - 1
exp(y) + -----------------------------------
           -1       -1            x - 1
          y   exp(y)   + ---------------------
                         - 2 exp(y) + O(x - 1)
```

Example 3. For rational functions, exact representations are returned when no specific 'number of terms' is requested. The method `"rational"` returns the rational expression equivalent to the continued fraction:

```
>> cf := contfrac((x - y)/(x^3 + y^3), x, 2):
   cf, contfrac::rational(cf);
```

```
                     -2       x       x - y
                  - y   + ---------, -----
                             3          3
                            y  + O(x)   y
```

```
>> cf := contfrac((x - y)/(x^3 + y^3), x):
   cf, contfrac::rational(cf);
```

```
       -2                  x                 x - y
    - y   + -----------------------------, -------
                             2              3    3
              3              x             x  + y
             y  + -------------------------
                     -1            x
                  - y   + -----------------
                              2        x
                          - y  + ----------
                                    -1
                                   y       x
                                   --- + ----
                                    2       2
                                         2 y
```

Example 4. The coefficients and expansion terms of a continued fraction can be accessed by the functions `nthcoeff` and `nthterm`:

```
>> cf := contfrac(sin(1/x), x = infinity, 4)
```

```
                          -1
                         x
                   --------------
                            -2
                           x
                   1 + ----------
                               -2
                       6 + O(x   )
```

```
>> nthcoeff(cf, 1), nthcoeff(cf, 2), nthcoeff(cf, 3), nthcoeff(cf, 4);
```

```
                    0, 1, 6, FAIL
```

```
>> nthterm(cf, 1), nthterm(cf, 2), nthterm(cf, 3)
```

```
                        1   1
                        -, --, FAIL
                        x   2
                           x
```

```
>> delete cf:
```

Example 5. We can compute a series expansion of a continued fraction via `series`:

```
>> cf := contfrac(sin(x)/(x - PI) - 1, x = PI)
```

```
                                  2
                          (x - PI)
             - 2 + -----------------------
                                     2
                            (x - PI)
                   6 + -------------------
                                         2
                       10/3 + O((x - PI) )
```

If no further arguments are given in `series`, the default expansion variable is `op`(cf, 3); the default expansion point is `op`(cf, 4):

```
>> op(cf, 3), op(cf, 4)
```

```
                        x, PI
```

```
>> series(cf)
```

```
                       2           4
               (x - PI)    (x - PI)               6
         - 2 + --------- - --------- + O((x - PI) )
                   6          120
```

Both the series variable as well as the expansion point may be passed explicitly to `series`.

```
>> series(cf, x = PI)
```

```
                       2           4
               (x - PI)    (x - PI)               6
         - 2 + --------- - --------- + O((x - PI) )
                   6          120
```

However, the values must coincide with the values used to compute the continued fraction: In the following call, the default expansion point `x = 0` is used by `series`. This clashes with the expansion point `x = PI` of the continued fraction:

```
>> series(cf, x)

 Error: 2nd argument: the expansion point PI of the continued f\
 raction clashes with the requested expansion point 0 [contfrac\
 ::series]

>> delete cf:
```

Changes:

- `contfrac` is a new function.

`debug` – execute a procedure in single-step mode

`debug(statement)` starts the MuPAD debugger, allowing to execute `statement` step by step.

Call(s):

- `debug()`
- `debug(statement)`

Parameters:

`statement` — any MuPAD object; typically a function call

Return Value: the return value of `statement` or `TRUE` or `FALSE`.

Related Functions: `noDebug`, `Pref::ignoreNoDebug`, `prog::check`, `prog::profile`, `prog::trace`

Details:

- `debug` called with an argument switches the state of the MuPAD kernel to *debug mode* and, if `statement` contains procedure calls that can be debugged, enters the interactive MuPAD debugger for controlled single-step execution of `statement`.
- If `debug` is called without arguments, the current state is returned *without* changing the state. If the debugger is on, the return value is `TRUE`, otherwise `FALSE`.

- In a MuPAD version with a graphical user interface, a separate debugger window pops up. In the UNIX terminal version, the text interface of the command line debugger is activated.

  The debugger features single stepping, inspection of variables and stack frames, breakpoints, etc. Read the online help of the debugger window for a description.

- Debugging is possible only for procedures written in the MuPAD language that do not have the option *noDebug*. In particular, debugging of kernel functions is not possible.

  After calling `Pref::ignoreNoDebug(TRUE)`, the procedure option *noDebug* is ignored.

- You can also debug a sequence of statements separated by semicolons if the sequence is enclosed in parentheses.

- `debug(statement)` returns the same result as `statement`, if the execution is not aborted within the debugger by the user.

- `debug` is a function of the system kernel.

Example 1. To proove, whether the kernel is in debug mode, `debug()` is called:

```
>> debug()
```

```
FALSE
```

To switch on the debugger mode, `debug(1)` is called:

```
>> debug(1)
```

```
Activating debugger...

For those library functions which are already loaded, the format of
the source code  displayed by the debugger  may differ from that of
the original source code file. To avoid this, restart the kernel in
debug mode.
Execution completed.
```

```
1
```

```
>> debug()
```

```
TRUE
```

Example 2. We start the debugger for stepwise execution of the statement `int(cos(x),x)`, which integrates the cosine function:

```
>> debug(int(cos(x), x)):
```

Background:

- In debug mode, the MuPAD parser is re-configured. When a procedure is read from a file, the parser inserts additional *debug nodes* containing file identifications and line numbers into procedures. These debug nodes allow the debugger to associate the currently executed piece of MuPAD code with the corresponding source text file.

- If the debug mode is activated and MuPAD encounters a procedure without debug nodes, it will write the procedure to a temporary file and add debug nodes on the fly. This allows interactively entered procedures to be debugged in the same way as procedures read from files. The temporary debug file is deleted at the end of the session.

  Since this also applies to procedures that were read before debug mode was switched on, it is recommended to start the kernel in debug mode (see below) when bigger applications are to be debugged.

- If the MuPAD kernel was not started in debug mode, this mode is turned on at the first execution of `debug`. It remains activated until the end of the session.

  It is possible to start the kernel in debug mode. On Windows platforms, this can be configured by choosing "Options" in the "View" menu and then clicking on "Kernel". In the graphical user interface on UNIX systems, clicking on "Kernel Debug Mode" in the "Options" menu toggles this setting. On a Macintosh, choose "Preferences" from the "File" menu and then "Kernel".

Changes:

- A call of `debug` without arguments returns the state of the debugger via `TRUE` or `FALSE`.

`degree` – the degree of a polynomial

`degree(p)` returns the total degree of the polynomial `p`.

`degree(p, x)` returns the degree of `p` with respect to the variable `x`.

Call(s):

- degree(p)
- degree(p, x)
- degree(f <, vars>)
- degree(f <, vars>, x)

Parameters:

p	—	a polynomial of type DOM_POLY
f	—	a polynomial expression
vars	—	a list of indeterminates of the polynomial: typically, identifiers or indexed identifiers
x	—	an indeterminate

Return Value: a nonnegative number. FAIL is returned if the input cannot be converted to a polynomial.

Overloadable by: p, f

Related Functions: coeff, degreevec, ground, lcoeff, ldegree, lmonomial, lterm, nterms, nthcoeff, nthmonomial, nthterm, poly, poly2list, tcoeff

---

Details:

- If the first argument f is not element of a polynomial domain, then degree converts the expression internally to a polynomial of type DOM_POLY via poly(f). If a list of indeterminates is specified, the polynomial poly(f, vars) is considered.
- degree(f, vars, x) returns 0 if x is not an element of the list vars.
- The degree of the zero polynomial is defined as 0.
- degree is a function of the system kernel.

Example 1. The total degree of the terms in the following polynomial expression is computed:

```
>> degree(x^3 + x^2*y^2 + 2)

                                   4
```

Example 2. `degree` may be applied to polynomials of type `DOM_POLY`:

```
>> degree(poly(x^2*z + x*z^3 + 1, [x, z]))

                                    4
```

Example 3. The next expression is regarded as a bi-variate polynomial in `x` and `z`. The degree with respect to `z` is computed:

```
>> degree(x^2*z + x*z^3 + 1, [x, z], z)

                                    3
```

Example 4. The degree of the zero polynomial is defined as 0:

```
>> degree(0, [x, y])

                                    0
```

`degreevec` – the exponents of the leading term of a polynomial

`degreevec(p)` returns a list with the exponents of the leading term of the polynomial `p`.

Call(s):

- `degreevec(p <, order>)`
- `degreevec(f <, vars> <, order>)`

Parameters:

`p`	—	a polynomial of type `DOM_POLY`
`order`	—	the term ordering: either *`LexOrder`*, or *`DegreeOrder`*, or *`DegInvLexOrder`*, or a user-defined term ordering of type `Dom::MonomOrdering`. The default is the lexicographical ordering *`LexOrder`*.
`f`	—	a polynomial expression
`vars`	—	a list of indeterminates of the polynomial: typically, identifiers or indexed identifiers

Return Value: a list of nonnegative integers. `FAIL` is returned if the input cannot be converted to a polynomial.

Overloadable by: `p, f`

Related Functions: `coeff`, `degree`, `ground`, `lcoeff`, `ldegree`, `lmonomial`, `lterm`, `nterms`, `nthcoeff`, `nthmonomial`, `nthterm`, `poly`, `poly2list`, `tcoeff`

Details:

- If the first argument `f` is not element of a polynomial domain, then `degreevec` converts the expression internally to a polynomial of type `DOM_POLY` via `poly(f)`. If a list of indeterminates is specified, the polynomial `poly(f, vars)` is considered.
- For a polynomial in the variables $x_1, x_2, \ldots, x_n$ with the leading term $x_1^{e_1} \times x_2^{e_2} \times \cdots \times x_n^{e_n}$, the exponent vector $[e_1, e_2, \ldots, e_n]$ is returned.
- `degreevec` returns a list of zeroes for the zero polynomial.
- For the orderings *`LexOrder`*, *`DegreeOrder`* and *`DegInvLexOrder`*, the result is computed by a fast kernel function. Other orderings are handled by slower library functions.

Example 1. The leading term of the following polynomial expression (with respect to the main variable `x`) is x^4:

```
>> degreevec(x^4 + x^2*y^3 + 2, [x, y])

                    [4, 0]
```

With the main variable `y`, the leading term is x^2y^3:

```
>> degreevec(x^4 + x^2*y^3 + 2, [y, x])

                    [3, 2]
```

For polynomials of type `DOM_POLY`, the indeterminates are an integral part of the data type:

```
>> degreevec(poly(x^4 + x^2*y^3 + 2, [x, y])),
   degreevec(poly(x^4 + x^2*y^3 + 2, [y, x]))

                 [4, 0], [3, 2]
```

Example 2. For a univariate polynomial, the standard term orderings regard the same term as "leading":

```
>> degreevec(poly(x^2*z + x*z^3 + 1, [x]), LexOrder),
   degreevec(poly(x^2*z + x*z^3 + 1, [x]), DegreeOrder),
   degreevec(poly(x^2*z + x*z^3 + 1, [x]), DegInvLexOrder)

                        [2], [2], [2]
```

In the multivariate case, different polynomial orderings may yield different leading exponent vectors:

```
>> degreevec(poly(x^2*z + x*z^3 + 1, [x, z])),
   degreevec(poly(x^2*z + x*z^3 + 1, [x, z]), DegreeOrder)

                        [2, 1], [1, 3]
```

```
>> degreevec(x^3 + x*y^2*z - 5*y^4, [x, y, z], LexOrder),
   degreevec(x^3 + x*y^2*z - 5*y^4, [x, y, z], DegreeOrder),
   degreevec(x^3 + x*y^2*z - 5*y^4, [x, y, z], DegInvLexOrder)

              [3, 0, 0], [1, 2, 1], [0, 4, 0]
```

Example 3. The exponent vector of the zero polynomial is a list of zeroes:

```
>> degreevec(0, [x, y, z])

                          [0, 0, 0]
```

`delete` – delete the value of an identifier

The statement `delete x` deletes the value of the identifier `x`.

Call(s):

- `delete x1, x2, ...`
- `_delete(x1, x2, ...)`

Parameters:

`x1, x2, ...` — identifiers or indexed identifiers

Return Value: the void object of type `DOM_NULL`.

Related Functions: `:=`, `_assign`, `assign`, `assignElements`, `evalassign`

Details:

- For many computations, symbolic variables are needed. E.g., solving an equation for an unknown `x` requires an identifier `x` that does not have a value. If `x` has a value, the statement `delete x` deletes the value and `x` can be used as a symbolic variable.
- The statement `delete x1, x2, ...` is equivalent to the function call `_delete(x1, x2, ...)`. The values of all specified identifiers are deleted.
- The statement `delete x[j]` deletes the entry `j` of a list, an array, or a table named `x`. Deletion of elements or entries reduces the size of lists and tables, respectively.
- If `A` is a `frame`, the statement `delete A::x` deletes the value of the identifier `x` in the frame `A`, leaving `x` as a *symbol* in that frame. Further information can be found on the `frame` help page.
- If `x` is an identifier carrying properties set via `assume`, then `delete x` detaches all properties from `x`, i.e., `delete x` has the same effect as `unassume(x)`. Cf. example 3.
- `_delete` is a function of the system kernel.

Example 1. The identifiers `x`, `y` are assigned values. After deletion, the identifiers have no values any longer:

```
>> x := 42: y := 7: delete x: x, y

                    x, 7

>> delete y: x, y

                    x, y
```

More than one identifier can be deleted by one call:

```
>> a := b := c := 42: a, b, c

                    42, 42, 42

>> delete a, b, c: a, b, c

                    a, b, c
```

Example 2. `delete` can also be used to delete specific elements of lists, arrays, and tables:

```
>> L := [7, 13, 42]

                              [7, 13, 42]

>> delete L[2]: L

                                [7, 42]

>> A := array(1..3, [7, 13, 42])

                            +-         -+
                            | 7, 13, 42 |
                            +-         -+

>> delete A[2]: A, A[2]

                        +-            -+
                        | 7, ?[2], 42 |, A[2]
                        +-            -+

>> T := table(1 = 7, 2 = 13, 3 = 42)

                              table(
                                3 = 42,
                                2 = 13,
                                1 = 7
                              )

>> delete T[2]: T

                              table(
                                3 = 42,
                                1 = 7
                              )
```

Note that `delete` does not evaluate the objects that are to be deleted. In the following, an element of the list `U` is deleted. The original value of `U` (the list `L`) is not changed:

```
>> U := L: delete U[1]: U, L

                            [42], [7, 42]
```

Finally, all assigned values are deleted:

```
>> delete U, L, A, T: U, L, A, T

                             U, L, A, T
```

Example 3. `delete` can also be used to delete properties of identifiers set via `assume`. With the assumption '`x > 1`', the expression `ln(x)` hat the property '`ln(x) > 0`', i.e., its sign is 1:

```
>> assume(x > 1): sign(ln(x))

                                   1
```

Without a property of `x`, the function `sign` cannot determine the sign of `ln(x)`:

```
>> delete x: sign(ln(x))

                               sign(ln(x))
```

`denom` – the denominator of a rational expression

`denom(f)` returns the denominator of the expression `f`.

Call(s):

- `denom(f)`

Parameters:

`f` — an arithmetical expression

Return Value: an arithmetical expression.

Overloadable by: `f`

Related Functions: `gcd`, `factor`, `normal`, `numer`

Details:

- `denom` regards the input as a rational expression: non-rational subexpressions such as `sin(x)`, `x^(1/2)` etc. are internally replaced by "temporary variables". The denominator of this rationalized expression is computed, the temporary variables are finally replaced by the original subexpressions.
- Numerator and denominator are not necessarily cancelled: the denominator returned by `denom` may have a non-trivial `gcd` with the numerator returned by `numer`. Pre-process the expression by `normal` to enforce cancellation of common factors. Cf. example 2.

NOTE

Example 1. We compute the denominators of some expressions:

```
>> denom(-3/4)

                              4

>> denom(x + 1/(2/3*x -2/x))

                               2
                            2 x  - 6

>> denom((cos(x)^2 -1)/(cos(x) -1))

                           cos(x) - 1
```

Example 2. `denom` performs no cancellations if the rational expression is of the form "numerator/denominator":

```
>> r := (x^2 - 1)/(x^3 - x^2 + x - 1): denom(r)

                              2    3
                         x - x  + x  - 1
```

This denominator has a common factor with the numerator of `r`; `normal` enforces cancellation of common factors:

```
>> denom(normal(r))

                              2
                             x  + 1
```

However, automatic normalization occurs if the input expression is a sum:

```
>> denom(r + x/(x + 1) + 1/(x + 1) - 1)

                              2
                             x  + 1

>> delete r:
```

`diff` – differentiate an expression or a polynomial

`diff(f, x)` computes the (partial) derivative $\partial f/\partial x$ of the function `f` with respect to the variable `x`.

Call(s):

- `diff(f)`
- `diff(f, x)`
- `diff(f, x1, x2, ...)`

Parameters:

`f`	—	an arithmetical expression or a polynomial of type `DOM_POLY`
`x, x1, x2, ...`	—	indeterminates: identifiers or indexed identifiers

Return Value: an arithmetical expression or a polynomial.

Overloadable by: `f`

Further Documentation: Section 7.1 of the MuPAD Tutorial.

Related Functions: `D`, `int`, `limit`, `poly`, `taylor`

Details:

- `diff(f, x)` computes the derivative of the arithmetical expression (or polynomial) `f` with respect to the indeterminate `x`.
- `diff(f, x1, x2, ...)` is equivalent to `diff(...diff(diff(f, x1), x2)...)`, i.e., the system first differentiates `f` with respect to `x1`, then differentiates the result with respect to `x2`, and so on, i.e., it computes the partial derivative $\cdots \frac{\partial}{\partial x_2} \frac{\partial}{\partial x_1} f$. Cf. example 3. In fact, the system internally converts nested `diff` calls into a single `diff` call with multiple arguments. Cf. example 7.
- `diff(f)` returns its evaluated argument: the "zero-th" derivative of `f` is `f` itself.
- It is convenient to compute higher derivatives using the sequence operator: If `n` is a nonnegative integer, then `diff(f, x $ n)` returns the n-th derivative of `f` with respect to `x`. Cf. example 4.
- The indeterminates `x, x1, x2, ...` must be either identifiers (of domain type `DOM_IDENT`) or indexed identifiers, i.e., of the form `x[n]`, where `x` is an identifier and `n` is an integer. If one of them is of a different form, then a symbolic `diff` call is returned. Cf. example 2.
- If `f` is an arithmetical expression, then `diff` returns an arithmetical expression. If `f` is a polynomial, then `diff` returns a polynomial as well. Cf. example 5. An exception to these rules occurs when the system is unable to compute the derivative, in which case it returns a symbolic `diff` call. Cf. example 6.

- MuPAD assumes that partial derivatives with respect to different indeterminates commute, i.e., `diff(f, x1, x2)` and `diff(f, x2, x1)` produce the same result `diff(f, y1, y2)`, where `[y1, y2]` = `sort([x1, x2])`. Cf. example 8.
- Users can extend the functionality of `diff` for their own special mathematical functions via overloading. This works by turning the corresponding function into a function environment and implementing the derivation rule for the function as the `"diff" slot` of the function environment. Cf. example 11.
- MuPAD has two functions for differentiation: `diff` and `D`. `D` represents the differential operator that may be applied to *functions*; `diff` is used to differentiate *arithmetical expressions*. Mathematically, `D(f)(x)` coincides with `diff(f(x), x)`; `D([1, 2], f)(x, y)` coincides with `diff(f(x, y), x, y)`. Symbolic calls of `D` and `diff` can be converted to one another via `rewrite`. Cf. example 10.
- `diff` is a function of the system kernel.

Example 1. We compute the derivative of x^2 with respect to `x`:

```
>> diff(x^2, x)

                              2 x
```

Example 2. You can differentiate with respect to an indexed identifier if the index is an integer:

```
>> diff(x[1]*y + x[1]*x[r], x[1])

                            y + x[r]
```

If the index is not an integer, then a symbolic `diff` call is returned:

```
>> diff(x[1]*y + x[1]*x[r], x[r])

                diff(y x[1] + x[r] x[1], x[r])
```

Example 3. You can differentiate with respect to more than one variable with a single `diff` call. In the following example, we differentiate first with respect to `x` and then with respect to `y`:

```
>> diff(x^2*sin(y), x, y) = diff(diff(x^2*sin(y), x), y)

                    2 x cos(y) = 2 x cos(y)
```

Example 4. We use the sequence operator `$` to compute the third derivative of the following expression with respect to `x`:

```
>> diff(sin(x)*cos(x), x $ 3)

                          2            2
                4 sin(x)  - 4 cos(x)
```

Example 5. `Polynomials` may be differentiated with respect to both the polynomial indeterminates (in the example below: `x`) or the parameters in the coefficients (in the example below: `a`):

```
>> diff(poly(sin(a)*x^3 + 2*x, [x]), x)

                                 2
            poly((3 sin(a)) x  + 2, [x])

>> diff(poly(sin(a)*x^3 + 2*x, [x]), a)

                                 3
                poly(cos(a) x , [x])
```

Example 6. The system returns the derivative of an unknown function as a symbolic `diff` call:

```
>> diff(f(x) + x, x)

                  diff(f(x), x) + 1
```

Example 7. The system internally converts nested `diff` calls into a single `diff` call with multiple arguments:

```
>> diff(diff(f(x, y), x), y)

                 diff(f(x, y), x, y)
```

Example 8. Partial derivatives with respect to several indeterminates are rewritten to a "normalized" ordering:

```
>> diff(f(x, y), x, y) = diff(f(x, y), y, x);

      diff(f(x, y), x, y) = diff(f(x, y), x, y)
```

Example 9. `diff` knows how to differentiate symbolic integrals:

```
>> int(f(x), x)

                        int(f(x), x)

>> diff(%, x, x)

                        diff(f(x), x)

>> int(f(t, x), t = x..x^2)

                                              2
                   int(f(t, x), t = x..x )

>> diff(%, x)

            2                                               2
    2 x f(x  , x) - f(x, x) + int(diff(f(t, x), x), t = x..x )
```

Example 10. `D` may only be applied to functions whereas `diff` is applied to expressions:

```
>> D(sin), diff(sin(x), x)

                         cos, cos(x)
```

Applying `D` to expressions and `diff` to functions makes no sense:

```
>> D(sin(x)), diff(sin, x)

                        D(sin(x)), 0
```

`rewrite` allows to rewrite expressions with `D` into `diff`-expressions and vice versa:

```
>> rewrite(D(f)(x), diff), rewrite(D(D(f))(x), diff)

                 diff(f(x), x), diff(f(x), x, x)

>> diff(f(x, x), x) = rewrite(diff(f(x, x), x), D)

      diff(f(x, x), x) = D([1], f)(x, x) + D([2], f)(x, x)
```

Example 11. Advanced users can extend `diff` to their own special mathematical functions (see the section "Background" below). To this end, embed your mathematical function into a function environment `g`, say, and implement the behavior of `diff` for this function as the `"diff"` slot of the function environment.

If a subexpression of the form `g(..)` occurs in an expression `f`, then `diff(f, x)` calls `g::diff(g(..), x)` to determine the derivative of the subexpression `g(..)`.

For illustration, we show how this works for the exponential function. Of course, the function environment `exp` already has a `"diff"` slot. We call our function environment `Exp` in order not to overwrite the existing system function `exp`.

This example `"diff"`-slot implements the chain rule for the exponential function. The derivative is the original function call times the derivative of the argument:

```
>> Exp := funcenv(Exp):
   Exp::diff := proc(f, x)
   begin
      // f = Exp(something), i.e., something = op(f, 1)
      f*diff(op(f, 1), x):
   end_proc:
   diff(Exp(x^2), x)

                                    2
                          2 x Exp(x )
```

In the following call, `prog::trace` shows that the function is called with only two arguments. `Exp::diff` is called only once since the result of the second call is read from an internal cache for intermediate results in `diff`:

```
>> prog::trace(Exp::diff):
   diff(Exp(x^2), x, x)

 enter 'Exp::diff'                       with args   : Exp(x^2), x
 leave 'Exp::diff'                       with result : 2*x*Exp(x^2)

                          2        2      2
                   2 Exp(x ) + 4 x  Exp(x )

>> prog::untrace(Exp::diff):  delete f, Exp:
```

Background:

- If a subexpression of the form `g(..)` occurs in `f` and `g` is a function environment, then `diff(f, x)` attempts to call the slot `"diff"` of `g` to determine the derivative of `g(..)`. In this way, you can extend the functionality of `diff` to your own special mathematical functions.

The slot "diff" is called with the arguments `g(..), x`.

If `g` does not have a slot "diff", then the system function `diff` returns the symbolic expression `diff(g(..), x)` for the derivative of the subexpression.

The "diff"-slot is always called with exactly two arguments. If the function `diff` was called with more indeterminates (i.e., if a higher derivative was requested), then the "diff"-slot is called several times, each call computing the derivative with respect to one of the indeterminates. The results of the calls of "diff"-slots are cached in `diff` in order to prevent redundant function calls. Cf. example 11.

- Similarly, if an element `d` of a library domain `T` occurs as a subexpression of `f`, then `diff(f, x)` calls the slot `T::diff(d, x)` to compute the derivative of `d`.

  If the domain `T` does not have a slot "diff", then `diff` considers this object as a constant and returns 0 for the corresponding subexpression.

Changes:

- Assuming that partial derivatives commute, the ordering of higher partial derivatives is now "normalized": `diff(f(x1, x2), x1, x2)` and `diff(f(x1, x2), x2, x1)` now produce the same result.

`dilog` – the dilogarithm function

`dilog(x)` represents the dilogarithm function $\int_1^x \ln(t)/(1-t)\,\mathrm{d}t$.

Call(s):

- `dilog(x)`

Parameters:

`x` — an arithmetical expression

Return Value: an arithmetical expression.

Overloadable by: `x`

Side Effects: When called with a floating point argument, the function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Related Functions: `ln`, `polylog`

Details:

- If `x` is a floating point number, then `dilog(x)` returns the numerical value of the dilogarithm function. The special values:

  `dilog(-1)` $= \pi^2/4 - i\,\pi\,\ln(2)$,

  `dilog(0)` $= \pi^2/6$,

  `dilog(1/2)` $= \pi^2/12 - \ln(2)^2/2$,

  `dilog(1)` $= 0$,

  `dilog(2)` $= -\pi^2/12$,

  `dilog(I)` $= \pi^2/16 - i$ `CATALAN` $- i\,\pi\,\ln(2)/4$,

  `dilog(-I)`$= \pi^2/16 + i$ `CATALAN` $+ i\,\pi\,\ln(2)/4$,

  `dilog(1+I)` $= -\pi^2/48 - i$ `CATALAN`,

  `dilog(1-I)` $= -\pi^2/48 + i$ `CATALAN`,

  `dilog(infinity)` = `-infinity`

  are implemented. For all other arguments, `dilog` returns a symbolic function call.

- Functional identities are used to rewrite the result for exact numerical arguments of `Type::Numeric` that have a negative real part or are of absolute value larger than 1. Cf. example 2.

- `dilog(x)` coincides with `polylog(2, 1-x)`.

Example 1. We demonstrate some calls with exact and symbolic input data:

```
>> dilog(0), dilog(2/3), dilog(sqrt(2)), dilog(1 + I), dilog(x)

      2                                          2
    PI                       1/2               PI
    ---, dilog(2/3), dilog(2   ), - I CATALAN - ---, dilog(x)
     6                                          48
```

Floating point values are computed for floating point arguments:

```
>> dilog(-1.2), dilog(3.4 - 5.6*I)

    2.458586602 - 2.477011851 I, - 2.529187195 + 2.25273709 I
```

Example 2. Arguments built from integers and rational numbers are rewritten, if they lie in the left half of the complex plane or are of absolute value larger than 1. The following arguments have a negative real part:

```
>> dilog(-400/3), dilog(-1/2 + I)

   2                            2
 PI                   ln(403/3)
 --- + dilog(3/403) + ---------- - ln(403/3) (I PI + ln(400/3))
  6                       2

        2              2
      PI    ln(3/2 - I)
    , --- + ----------- + dilog(6/13 + 4/13 I) -
       6         2

    ln(- 1/2 + I) ln(3/2 - I)
```

The following arguments have an absolute value larger than 1:

```
>> dilog(31/30), dilog(1 + 2/3*I)

                           2                 2
                  ln(31/30)     ln(1 + 2/3 I)
 - dilog(30/31) - ----------, - -------------- -
                      2               2

    dilog(9/13 - 6/13 I)
```

Example 3. The negative real axis is a branch cut of `dilog`. A jump of height $2\pi i \ln(1-x)$ occurs when crossing this cut at the real point $x < 0$:

```
>> dilog(-1.2), dilog(-1.2 + I/10^100), dilog(-1.2 - I/10^100)

 2.458586602 - 2.477011851 I, 2.458586602 - 2.477011851 I,

    2.458586602 + 2.477011851 I
```

Example 4. The functions `diff`, `float`, `limit`, and `series` handle expressions involving `dilog`:

```
>> diff(dilog(x), x, x, x), float(ln(3 + dilog(sqrt(PI))))

         2 ln(x)        2            1
        -------- + ---------- - ----------, 0.8503829845
               3            2    2
        (1 - x)    x (1 - x)    x  (1 - x)
```

```
>> limit(dilog(x^10 + 1)/x, x = infinity)

                                0

>> series(dilog(x + 1/x)/x, x = -infinity, 3)

     2                  2
   PI    (I PI + ln(-x))
 - --- - ----------------
    6           2         I PI + ln(-x) + 1
 ------------------------ + ----------------- +
            x                       2
                                   x

                ln(-x)
    - 1/2 I PI - ------ + 1/4
                   2          /  1 \
    ------------------------ + O| -- |
                3             |  4 |
               x              \ x  /
```

Background:

- dilog(x) coincides with $\sum_{k=1}^{\infty}(1-x)^k/k^2$ for $|x|<1$.
- dilog has a branch cut along the negative real axis. The value at a point x on the cut coincides with the limit "from above":

$$\mathtt{dilog}(x) = \lim_{\epsilon\to 0_+} \mathtt{dilog}(x+\epsilon\, i) = \lim_{\epsilon\to 0_-} \mathtt{dilog}(x+\epsilon\, i) - 2\,\pi\, i\, \ln(1-x)\ .$$

- Reference: L. Lewin (ed.), "Structural Properties of Polylogarithms", Mathematical Surveys and Monographs Vol. 37, American Mathematical Society, Providence (1991).

---

dirac – the Dirac delta distribution

dirac(x) represents the Dirac delta distribution.

dirac(x, n) represents the n-th derivative of the delta distribution.

Call(s):

- dirac(x)
- dirac(x, n)

Parameters:

x — an arithmetical expression
n — an arithmetical expression representing a nonnegative integer

Return Value: an arithmetical expression.

Overloadable by: `x`

Side Effects: `dirac` reacts to properties of identifiers.

Related Functions: `heaviside`

Details:

- The calls `dirac(x, 0)` and `dirac(x)` are equivalent.
- If the argument `x` represents a non-zero real number, then 0 is returned. If `x` is a non-real number of domain type `DOM_COMPLEX`, then `undefined` is returned. For all other arguments, a symbolic function call is returned.
- `dirac` does not have a predefined value at the origin. Use

  `unprotect(dirac): dirac(0) := myValue:`

  and

  `dirac(float(0)) := myFloatValue: protect(dirac):`

  to assign a value (e.g., `infinity`).
- For univariate linear expressions, the simplification rule

$$\delta^{(n)}(a\,x - b) = \frac{\mathtt{sign}(a)}{a^{n+1}}\,\delta^{(n)}\Big(x - \frac{b}{a}\Big)$$

  is implemented for real numerical values `a`.
- The integration function `int` treats `dirac` as the usual delta distribution. Cf. example 3.

Example 1. `dirac` returns 0 for arguments representing non-zero real numbers:

```
>> dirac(-3), dirac(3/2), dirac(2.1, 1),
   dirac(3*PI), dirac(sqrt(3), 3)

                    0, 0, 0, 0, 0
```

Arguments of domain type `DOM_COMPLEX` yield `undefined`:

```
>> dirac(1 + I), dirac(2/3 + 7*I), dirac(0.1*I, 1)
```

```
undefined, undefined, undefined
```

A symbolic call is returned for other arguments:

```
>> dirac(0), dirac(x), dirac(ln(-5)), dirac(x + I, 2), dirac(x, n)

 dirac(0), dirac(x), dirac(I PI + ln(5)), dirac(x + I, 2),

    dirac(x, n)
```

```
>> dirac(2*x - 1, n)

                        dirac(x - 1/2, n)
                        -----------------
                               n + 1
                              2
```

A natural value for `dirac(0)` is `infinity`:

```
>> unprotect(dirac): dirac(0) := infinity: dirac(0)

                            infinity
```

```
>> delete dirac(0): protect(dirac): dirac(0)

                            dirac(0)
```

Example 2. `dirac` reacts to assumptions set by `assume`:

```
>> assume(x < 0): dirac(x)

                                0
```

```
>> assume(x, Type::Real): assume(x <> 0, _and): dirac(x)

                                0
```

```
>> unassume(x):
```

Example 3. The symbolic integration function `int` treats `dirac` as the delta distribution:

```
>> int(f(x)*dirac(x - y^2), x = -infinity..infinity)

                                 2
                              f(y )
```

```
>> int(int(f(x, y)*dirac(x - y^2), x = -infinity..infinity),
       y = -1..1)

                                 2
                        int(f(y  , y), y = -1..1)
```

The indefinite integral of `dirac` involves the step function `heaviside`:

```
>> int(f(x)*dirac(x), x), int(f(x)*dirac(x, 1), x)

     heaviside(x) f(0), dirac(x) f(0) - heaviside(x) D(f)(0)
```

If the delta peak is on the boundary of the integration region, then the result involves a symbolic call of `heaviside(0)`:

```
>> int(f(x)*dirac(x - 3), x = -1..3)

                          f(3) heaviside(0)
```

Note that `int` can handle the distribution only if the argument of `dirac` is linear in the integration variable:

```
>> int(f(x)*dirac(2*x - 3), x = -10..10),
   int(f(x)*dirac(x^2), x = -10..10)

            f(3/2)                 2
            ------, int(f(x) dirac(x ), x = -10..10)
               2
```

Also note that `dirac` should not be used for numerical integration, since the numerical algorithm will typically fail to detect the delta peak:

```
>> numeric::int(dirac(x - 3), x = -10..10)

                                 0.0
```

`discont` – discontinuities of a function

`discont(f, x)` computes the set of all discontinuities of the function $f(x)$.

`discont(f, x = a..b)` computes the set of all discontinuities of $f(x)$ lying in the interval $[a, b]$.

Call(s):

- `discont(f, x)`
- `discont(f, x, F)`
- `discont(f, x, *Undefined*)`
- `discont(f, x = a..b)`
- `discont(f, x = a..b, F)`
- `discont(f, x = a..b, *Undefined*)`

Parameters:

`f`	—	an arithmetical expression representing a function in `x`
`x`	—	an identifier
`F`	—	either `Dom::Real` or `Dom::Complex`
`a, b`	—	interval boundaries: arithmetical expressions

Options:

Undefined	—	return only those points where `f` is not defined (and not just discontinous).

Return Value: a set—see the help page for `solve` for an overview of all types of sets—or a symbolic `discont` call.

Side Effects: `discont` reacts to properties of free parameters both in `f` as well as in `a` and `b`. `discont` sometimes reacts to properties of `x`.

Overloadable by: `f`

Related Functions: `limit`, `solve`

Details:

- `discont(f, x, F)` returns a set of numbers containing all discontinuities of f when f is regarded as a function of x defined on F. Please note that a real number that is a discontinuity of a complex function need not be a discontinuity of the restriction of that function to the set of real numbers: consider, for example, a function that has its branch cut on the real axis, as in example 2 below.

- Discontinuities include points where the function is not defined as well as points where the function is defined but not continuous. If the option *Undefined* is used, only points where the function is not defined are returned.

- If the parameter `F` is omitted, then `F=Dom::Complex` is used as a default, i.e., `f` is regarded as a function defined on the complex numbers, unless the global assumption `assume(Global, Type::Real)` has been made, in which case `F=Dom::Real` is the default.

- If a range `a..b` is given, the set of discontinuities is intersected with the closed interval $[a, b]$.
- The set returned by `discont` may contain numbers that are not discontinuities of f. See example 7.
- If `discont` is unable to compute the discontinuities, then a symbolic `discont` call is returned; see example 8.
- `discont` can be extended to user-defined mathematical functions via overloading. To this end, embed the mathematical function in a function environment and assign the set of real and complex discontinuities and points where is the function is not defined to its `"realDiscont"`, `"complexDiscont"`, and `"undefined"` slot, respectively; see `solve` for an overview of the various types of sets. See also example 8 below.

Example 1. The `gamma` function has poles at all integers less or equal to zero. Hence `x -> gamma(x/2)` has poles at all even integers less or equal to zero:

```
>> discont(gamma(x/2), x)

          { 2*X2 |  X2 in Z_ } intersect ]-infinity, 0]
```

Example 2. The logarithm has a branch cut on the negative real axis; hence, it is not continuous there. However, its restriction to the real numbers is continuous at every point except zero:

```
>> discont(ln(x), x), discont(ln(x), x, Dom::Real)

                      ]-infinity, 0], {0}
```

Example 3. The function `sign` is defined everywhere; it is not continous at zero:

```
>> discont(sign(x), x), discont(sign(x), x, Undefined)

                            {0}, {}
```

Example 4. If a range is given, only the discontinuities in that range are returned.

```
>> discont(1/x/(x - 1), x = 0..1/2)

                              {0}
```

Example 5. A range may have arbitrary arithmetical expressions as boundaries. `discont` does not implicitly assume that the right boundary is greater or equal to the left boundary:

```
>> discont(1/x, x = a..b)

 piecewise({0} if a <= 0 and 0 <= b,

    {} if (not a <= 0 or not 0 <= b))
```

Example 6. As can be seen from the previous example, `discont` reacts to properties of free parameters (because `piecewise` does). The result also depends on the properties of `x`: it may omit values that `x` cannot take on anyway because of its properties.

```
>> assume(x > 0):
   discont(1/x, x)

                                 {}

>> delete x:
```

Example 7. Sometimes, `discont` returns a proper superset of the set of discontinuities:

```
>> discont(piecewise([x<>0, x*sin(1/x)], [x=0, 0]), x)

                                 {0}
```

Example 8. A symbolic `discont` call is returned if the system does not know how to determine the discontinuities of a given function:

```
>> delete f: discont(f(x), x)

                          discont(f(x), x)
```

You can provide the necessary information by adding a slot to `f`. `discont` takes care to handle `f` correctly also if it appears in a more complicated expression:

```
>> f := funcenv(x->procname(x)): f::complexDiscont:={1}:
   discont(f(sin(x)), x=-4..34)

             { PI  5 PI  9 PI  13 PI  17 PI  21 PI }
             { --, ----, ----, -----, -----, ----- }
             {  2    2     2     2      2      2   }
```

Example 9. We define a function that implements the logarithm to base 2. For simplicity, we let it always return the unevaluated function call. The logarithm has a branch cut on the negative real axis; its restriction to the reals is continuous everywhere except at zero:

```
>> binlog := funcenv(x -> procname(x)):
   binlog::realDiscont := {0}:
   binlog::undefined := {0}:
   binlog::complexDiscont := Dom::Interval(-infinity, [0]):
   discont(binlog(x), x=-2..2);
   discont(binlog(x), x=-2..2, Dom::Real);
   discont(binlog(x), x=-2..2, Undefined)
```

```
                         [-2, 0]

                           {0}

                           {0}
```

Changes:

- A new option ***Undefined*** has been introduced.

---

div – the integer part of a quotient

x div m represents the integer q satisfying $x = q\,m + r$ with $0 \leq r < |m|$.

Call(s):

- x div m
- _div(x, m)

Parameters:

x, m — integers or symbolic arithmetical expressions; m must not be zero.

Return Value: an integer or an arithmetical expression of type "_div".

Overloadable by: x, m

Related Functions: _mod, /, divide, mod, modp, mods

Details:

- For positive `x` and `m`, `q` = `x div m` is the integer part of the quotient `x/m`, i.e., `q` = `trunc(x/m)`.
- `x div m` is equivalent to the function call `_div(x, m)`.
- An integer is returned if both `x` and `m` evaluate to integers. A symbolic expression of type `"_div"` is returned if either `x` or `m` does not evaluate to a number. An error is raised if `x` or `m` evaluates to a number that is not an integer.
- `div` does not operate on polynomials. Use `divide`.
- `div` is a function of the system kernel.

Example 1. With the default setting for `mod`, the identity `(x div m) * m + (x mod m) = x` holds for integer numbers `x` and `m`:

```
>> 43 div 13 = trunc(43/13), 43 mod 13 = frac(43/13) * 13

                     3 = 3, 4 = 4

>> (43 div 13) * 13 + (43 mod 13) = 43

                        43 = 43
```

Example 2. Symbolic expressions of type `"_div"` are returned, if either `x` or `m` does not evaluate to a number:

```
>> 43 div m, x div 13, x div m

              43 div m, x div 13, x div m

>> type(x div m)

                         "_div"
```

If `x` or `m` are numbers, they must be integer numbers:

```
>> 1/2 div 2

 Error: Illegal argument in div or mod

>> x div 2.0

 Error: Illegal operand [_mod]
```

divide – **divide polynomials**

`divide(p, q)` divides the univariate polynomials `p` and `q`. It returns the quotient s and the remainder r satisfying $p = s\,q + r$, $\texttt{degree}(r) < \texttt{degree}(q)$.

Call(s):

- `divide(p1, q1 <, mode>)`
- `divide(f1, g1 <, mode>)`
- `divide(f, g <, [x]> <, mode>)`
- `divide(p, q,` *`Exact`*`)`
- `divide(f, g, <[x1, x2, ...],>` *`Exact`*`)`

Parameters:

`p1, q1`	—	univariate polynomials of type `DOM_POLY`.
`f1, g1`	—	univariate polynomial expressions
`p, q`	—	univariate or multivariate polynomials of type `DOM_POLY`.
`f, g`	—	univariate or multivariate polynomial expressions
`x`	—	an identifier or an indexed identifier. Expressions are regarded as univariate polynomials in the indeterminate `x`.
`x1, x2, ...`	—	identifiers or indexed identifiers. Multivariate expressions are regarded as multivariate polynomials in these indeterminates.

Options:

`mode`	—	either *`Quo`* or *`Rem`*. With *`Quo`*, only the quotient `s` is returned; with *`Rem`*, only the remainder `r` is returned.
`Exact`	—	exact division of multivariate polynomials. Only the quotient `s` is returned. If no exact division without remainder is possible, `FAIL` is returned.

Return Value: a polynomial, a polynomial expression, a sequence of two polynomials or polynomial expressions, or the value `FAIL`.

Overloadable by: `p, q, p1, q1, f, g, f1, g1`

Related Functions: `/`, `content`, `degree`, `div`, `factor`, `gcd`, `gcdex`, `groebner::normalf`, `ground`, `mod`, `multcoeffs`, `pdivide`, `poly`, `powermod`

Details:

- `divide(p, q)` divides the univariate polynomials `p` and `q`. The quotient `s` and the remainder `r` are calculated such that `p = s*q + r` and `degree(r) < degree(q)`. If no option is given, the sequence `s, r` is returned.
- The first two arguments can be either polynomials or polynomial expressions.

  Polynomials must be of the same type, i.e. their variables and coefficient rings must be identical.

  Expressions are internally converted to polynomials (see the function `poly`). If no list of indeterminates is specified, all symbolic variables in the expressions are chosen as indeterminates. `FAIL` is returned if the expressions cannot be converted to polynomials.

  The resulting polynomials are of the same type as the first two arguments, i.e., either polynomials of type `DOM_POLY` or polynomial expressions are returned.
- The coefficient ring of the polynomials must implement the method `"_divide"` which is used internally to divide coefficients. This method must return `FAIL` if coefficients cannot be divided.
- `divide` is a function of the system kernel.

Example 1. Without further options, `divide` returns the quotient and the remainder of the division of univariate polynomials:

```
>> divide(poly(x^3 + x + 1, [x]), poly(x^2 + x + 1, [x]))

               poly(x - 1, [x]), poly(x + 2, [x])

>> divide(x^3 + x + 1, x^2 + x + 1)

                         x - 1, x + 2
```

Example 2. If expressions contain more than one variable, indeterminates must be specified. Other symbolic objects are regarded as parameters. The option *Quo* instructs `divide` to return the quotient only:

```
>> divide(a*x^3 + x + 1, x^2 + x + 1, [x], Quo)

                            a x - a
```

The option *Rem* instructs `divide` to return the remainder only:

```
>> divide(a*x^3 + x + 1, x^2 + x + 1, [x], Rem)

                           a + x + 1
```

Example 3. For multivariate expressions, regarded as a univariate polynomial in a specified indeterminate, the result of the division depends on the indeterminate:

```
>> divide(x^2 - 2*x - y, y*x - 1, [x]);

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

>> divide(x^2 - 2*x - y, y*x - 1, [y])

                                 1   2   1
                               - -, x  - - - 2 x
                                 x       x
```

Example 4. Multivariate polynomials and polynomial expressions can only be divided with the option *Exact*. If a division without remainder is possible, the quotient is returned. This operation is equivalent to the division of polynomials using the / operator:

```
>> p := poly(x^2 - x*y - x + y, [x, y]): q := poly(x - 1, [x, y]):
   p/q = divide(p, q, Exact)

            poly(x - y, [x, y]) = poly(x - y, [x, y])
```

If exact division of multivariate polynomials without remainder is not possible, `FAIL` is returned:

```
>> p := poly(x^2 + y, [x, y]): q := poly(x - 1, [x, y]):
   divide(p, q, Exact) = p/q

                              FAIL = FAIL

>> delete p, q:
```

`domtype` – the data type of an object

`domtype(object)` returns the domain type (the data type) of the object.

Call(s):

- `domtype(object)`

Parameters:

`object` — any MuPAD object

Return Value: the data type, i.e., an object of type `DOM_DOMAIN`.

Overloadable by: `object`

Related Functions: `coerce`, `DOM_DOMAIN`, `domain`, `hastype`, `testtype`, `type`, `Type`

Details:

- For most data types, the domain type as returned by `domtype` coincides with the type returned by the function `type`. Only for expressions of domain type `DOM_EXPR`, the function `type` yields a distinction according to the 0-th operand. Cf. example 2.
- In contrast to most other functions, `domtype` does not flatten arguments that are expression sequences.
- `domtype` is a function of the system kernel.

Example 1. Real floating point numbers are of domain type `DOM_FLOAT`:

```
>> domtype(12.345)

                              DOM_FLOAT
```

Complex numbers are of domain type `DOM_COMPLEX`. The operands may be integers (`DOM_INT`), rational numbers (`DOM_RAT`), or floating point numbers (`DOM_FLOAT`). The operands can be accessed via `op`:

```
>> domtype(1 - 2*I), op(1 - 2*I);
   domtype(1/2 - I), op(1/2 - I);
   domtype(2.0 - 3.0*I), op(2.0 - 3.0*I)

                         DOM_COMPLEX, 1, -2

                        DOM_COMPLEX, 1/2, -1

                       DOM_COMPLEX, 2.0, -3.0
```

Example 2. Expressions are objects of the domain type `DOM_EXPR`. The type of expressions can be queried further with the function `type`:

```
>> domtype(x + y), type(x + y);
   domtype(x - 1.0*I), type(x - 1.0*I);
   domtype(x*I), type(x*I);
   domtype(x^y), type(x^y);
   domtype(x[i]), type(x[i])
```

```
                         DOM_EXPR, "_plus"

                         DOM_EXPR, "_plus"

                         DOM_EXPR, "_mult"

                        DOM_EXPR, "_power"

                        DOM_EXPR, "_index"
```

Example 3. `domtype` evaluates its argument. In this example, the assignment is first evaluated and `domtype` is applied to the return value of the assignment. This is the right hand side of the assignment, i.e., `5`:

```
>> domtype((a := 5))
```

```
                              DOM_INT
```

```
>> delete a:
```

Example 4. Here the identifier `a` is first evaluated to the `expression sequence 3, 4`. Its domain type is `DOM_EXPR`, its type is `"_exprseq"`:

```
>> a := 3, 4: domtype(a), type(a)
```

```
                       DOM_EXPR, "_exprseq"
```

```
>> delete a:
```

Example 5. `factor` creates objects of the domain type `Factored`:

```
>> domtype(factor(x^2 - x))
```

```
                             Factored
```

Example 6. `matrix` creates objects of the domain type `Dom::Matrix()`:

```
>> domtype(matrix([[1, 2], [3, 4]]))

                        Dom::Matrix()
```

Example 7. Domains are of the domain type `DOM_DOMAIN`:

```
>> domtype(DOM_INT), domtype(DOM_DOMAIN)

                    DOM_DOMAIN, DOM_DOMAIN
```

Example 8. `domtype` is overloadable, i.e., a domain can pretend to be of another domain type. The special `slot` `"dom"` always gives the actual domain:

```
>> d := newDomain("d"): d::domtype := x -> "domain type d":
   e := new(d, 1): e::dom, type(e), domtype(e)

                     d, d, "domain type d"

>> delete d, e:
```

---

end – close a block statement

`end` is a keyword which, depending on the context, is parsed as one of the following keywords:

- `end_case`
- `end_for`
- `end_if`
- `end_proc`
- `end_repeat`
- `end_while`

Related Functions: `end_case`, `end_for`, `end_if`, `end_proc`, `end_repeat`, `end_while`

Example 1. Each of the keywords `proc`, `case`, `if`, `for`, `repeat`, and `while` starts some block construct in the MuPAD language. Each block can be closed with `end` or with the corresponding special keyword `end_proc`, `end_case` etc.:

```
>> f :=
   proc(a, b)
     local i;
   begin
     for i from a to b do
       if isprime(i) then
          print(Unquoted, expr2text(i)." is a prime")
       end
     end
   end:
```

```
>> f(20, 30):
```

```
                          23 is a prime

                          29 is a prime
```

The parser translates `end` to the appropriate keyword matching the type of the block:

```
>> expose(f)
```

```
        proc(a, b)
          name f;
          local i;
        begin
          for i from a to b do
            if isprime(i) then
              print(Unquoted, expr2text(i)." is a prime")
            end_if
          end_for
        end_proc
```

```
>> delete f:
```

`erf`, `erfc` – the error function and the complementary error function

`erf(x)` represents the error function $\frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, \mathrm{d}t$. The complementary error function is $\mathtt{erfc}(x) = 1 - \mathtt{erf}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty e^{-t^2}\, \mathrm{d}t$.

Call(s):

- `erf(x)`
- `erfc(x)`

Parameters:

`x` — an arithmetical expression

Return Value: an arithmetical expression.

Side Effects: When called with a floating point argument, the functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Details:

- Theses functions are defined for all complex arguments.
- Floating point values are returned for floating point arguments. The exact values:

  $\texttt{erf}(0) = 0$, $\texttt{erf}(\texttt{infinity}) = 1$, $\texttt{erf}(-\texttt{infinity}) = -1$,

  $\texttt{erfc}(0) = 1$, $\texttt{erfc}(\texttt{infinity}) = 0$, $\texttt{erfc}(-\texttt{infinity}) = 2$

  are implemented. For all other arguments, symbolic function calls are returned.
- For floating point arguments of large absolute value, internal numerical underflow may happen. The section of the complex plane where $|\texttt{Im}(x)| \leq |\texttt{Re}(x)|/10$, is protected against such underflows: when the real part of x is a large positive number, the result returned by `erfc` may be truncated to `0.0`. For large negative real part, it may be rounded to `2.0`. Knowing that $\texttt{erf}(x) = 1 - \texttt{erfc}(x)$, `erf` may also return correspondingly rounded values for arguments in this section. Cf. example 2.
- The float attributes are kernel functions, i.e., floating point evaluation is fast.

Example 1. We demonstrate some calls with exact and symbolic input data:

```
>> erf(0), erf(3/2), erf(sqrt(2)), erf(infinity)

                                   1/2
                0, erf(3/2), erf(2    ), 1

>> erfc(0), erfc(x + 1), erfc(-infinity)

                    1, erfc(x + 1), 2
```

Floating point values are computed for floating point arguments:

```
>> erf(-7.2), erf(2.0 + 3.5*I), erfc(100.0 + 100.0*I)

 -1.0, 421.8123327 + 343.6612334 I,

    0.0006523436638 - 0.003935726363 I
```

Example 2. For large floating point arguments with positive real parts, the values returned by `erfc` may be truncated to `0.0`:

```
>> erfc(2411.3), erfc(2411.4)

                  3.678326052e-2525152, 0.0
```

This protection against numerical underflow is builtin for arguments satisfying $|\texttt{Im}(x)| \leq |\texttt{Re}(x)|/10$.

```
>> erfc(2500.0 + 250.0*I)

                              0.0
```

Errors may occur outside this region in the complex plane:

```
>> erfc(2500.0 + 250.1*I)

 Error: Overflow/underflow in arithmetical operation;
 during evaluation of 'erfc::float'
```

Example 3. The functions `diff`, `float`, `limit`, and `series` handle expressions involving the error functions:

```
>> diff(erf(x), x, x, x), float(ln(3 + erfc(sqrt(PI)*I)))

       2        2             2
    8 x  exp(- x )   4 exp(- x )
    -------------- - -----------, 2.309003461 - 1.16207002 I
          1/2            1/2
        PI             PI

>> limit(x/(1 + x)*erf(x), x = infinity)

                               1

>> series(erfc(x), x = infinity, 4)

           1                   1            /      1     \
     --------------- - ------------------ + O| ---------- |
        1/2      2        3   1/2      2     |  5      2  |
     x PI    exp(x )   2 x  PI    exp(x )    \ x  exp(x ) /
```

Background:

- `erf` and `erfc` are entire functions.

`error` – raise a user-specified exception

`error(message)` aborts the current procedure, returns to the interactive level, and displays the error message `message`.

Call(s):

- `error(message)`

Parameters:

`message` — the error message: a string

Side Effects: The formatting of the output of `error` is sensitive to the environment variable `TEXTWIDTH`.

Related Functions: `lasterror`, `prog::error`, `traperror`, `warning`

Details:

- The call `error(message)` aborts the current procedure with an error. If the error is not caught via `traperror` by a procedure that has directly or indirectly called the current procedure, control is returned to the interactive level, and the string `message` is printed as an error message.

- The printed error message has the form `Error: message [name]`, where `name` is the name of the procedure containing the call to `error`. See the examples.

- Errors can be caught by the function `traperror`. If an error occurs while the arguments of `traperror` are evaluated, control is returned to the procedure containing the call to `traperror` and not to the interactive level. No error message is printed. The return value of `traperror` is `1028` when it catches an error raised by `error`; see example 2.

- The function `error` is useful to raise an error in the type checking part of a user-defined procedure, when this procedure is called with invalid arguments.

- `error` is a function of the system kernel.

Example 1. If the divisor of the following simple division routine is `0`, then an error is raised:

```
>> mydivide := proc(n, d) begin
     if iszero(d) then
       error("Division by 0")
     end_if;
     n/d
   end_proc:
   mydivide(2, 0)

 Error: Division by 0 [mydivide]
```

Example 2. When the error is raised in the following procedure `p`, control is returned to the interactive level immediately. The second call to `print` is never executed. Note that the procedure's name is printed in the error message:

```
>> p := proc() begin
     print("entering procedure p");
     error("oops");
     print("leaving procedure p")
   end_proc:
   p()

                     "entering procedure p"
 Error: oops [p]
```

The following procedure `q` calls the procedure `p` and catches any error that is raised within `p`:

```
>> q := proc() begin
     print("entering procedure q");
     print("caught error: ", traperror(p()));
     print("leaving procedure q")
   end_proc:
   q()

                     "entering procedure q"

                     "entering procedure p"

                     "caught error: ", 1028

                      "leaving procedure q"
```

`eval` – evaluate an object

`eval(object)` evaluates its argument `object` by recursively replacing the identifiers occurring in it by their values and executing function calls, and then evaluates the result again.

Call(s):

- `eval(object)`

Parameters:

`object` — any MuPAD object

Return Value: the evaluated object.

Side Effects: `eval` is sensitive to the value of the environment variable `LEVEL`, which determines the maximal substitution depth for identifiers.

Further Documentation: Chapter 5 of the MuPAD Tutorial.

Related Functions: `context`, `evalassign`, `evalp`, `freeze`, `hold`, `indexval`, `LEVEL`, `level`, `MAXLEVEL`, `MAXDEPTH`, `val`

Details:

- `eval` serves to request the evaluation of unevaluated or partially evaluated objects. *Evaluation* means that identifiers are replaced by their values and function calls are executed.

  Usually, every system function automatically evaluates its arguments and returns a fully evaluated object, and using `eval` is only necessary in exceptional cases. For example, the functions `map`, `op`, and `subs` may return objects that are not fully evaluated. See example 1.

- Like most other MuPAD functions, `eval` first evaluates its argument. Then it evaluates the result again. At interactive level, the second evaluation usually has no effect, but this is different within procedures; see examples 3 and 4.

- `eval` is sensitive to the value of the environment variable `LEVEL`, which determines the maximal depth of the recursive process that replaces an identifier by its value during evaluation. The evaluation of the argument and the subsequent evaluation of the result both take place with substitution depth `LEVEL`. See example 3.

- If a local variable or a formal parameter, of type `DOM_VAR`, of a `procedure` occurs in `object`, then it is always replaced by its value when `eval` evaluates its argument, independent of the value of `LEVEL`. At the subsequent second evaluation, the value of the local variable is evaluated with substitution depth given by `LEVEL`, which usually is `1`. Cf. example 4.
- The behavior of `eval` within a procedure may sometimes not be what you expect, since the default substitution depth within procedures is `1` and `eval` evaluates with this substitution depth. Use `level` to request a complete evaluation within a procedure; see the corresponding help page for details.
- `eval` enforces the evaluation of expressions of the form `hold(x)`: `eval(hold(x))` is equivalent to `x`. Cf. example 2.
- `eval` accepts expression sequences as arguments; see example 3. In particular, the call `eval()` returns the empty sequence `null()`.
- `eval` does not recursively descend into `arrays`. Use the call `map(object, eval)` to evaluate the entries of an array. Cf. example 5.
- `eval` does not recursively descend into `tables`. Use the call `map(object, eval)` to evaluate the entries of a table.

  However, it is not possible to evaluate the indices of a given table. If you want to do this, create a new table with the evaluated operands of the old one. Cf. example 6.
- `Polynomials` are not further evaluated by `eval`. Use `evalp` to substitute values for the indeterminates of a polynomial, and use the call `mapcoeffs(object, eval)` to evaluate all coefficients. Cf. example 7.
- The evaluation of elements of a user-defined `domain` depends on the implementation of the domain. Usually, domain elements remain unevaluated. If the domain has a slot `"evaluate"`, the corresponding slot routine is called with the domain element as argument at each evaluation, and hence it is called twice when `eval` is invoked. Cf. example 8.
- `eval` is a function of the system kernel.

Example 1. `subs` performs a substitution, but does not evaluate the result:

```
>> subs(ln(x), x = 1)

                                ln(1)
```

An explicit call of `eval` is necessary to evaluate the result:

```
>> eval(subs(ln(x), x = 1))

                                  0
```

`text2expr` does not evaluate its result either:

```
>> a := c:
   text2expr("a + a"), eval(text2expr("a + a"))

                              a + a, 2 c
```

Example 2. The function `hold` prevents the evaluation of its argument. A later evaluation can be forced with `eval`:

```
>> hold(1 + 1);  eval(%)

                                 1 + 1

                                   2
```

Example 3. When an object is evaluated, identifiers are replaced by their values recursively. The maximal recursion depth of this process is given by the environment variable `LEVEL`:

```
>> delete a0, a1, a2, a3, a4:
   a0 := a1:  a1 := a2 + 2:  a2 := a3 + a4:  a3 := a4^2:  a4 := 5:

>> LEVEL := 1:  a0, a0 + a2;
   LEVEL := 2:  a0, a0 + a2;
   LEVEL := 3:  a0, a0 + a2;
   LEVEL := 4:  a0, a0 + a2;
   LEVEL := 5:  a0, a0 + a2;

                            a1, a1 + a3 + a4

                                       2
                          a2 + 2, a2 + a4  + 7

                        a3 + a4 + 2, a3 + a4 + 32

                               2        2
                            a4  + 7, a4  + 37

                                 32, 62
```

`eval` first evaluates its argument and then evaluates the result again. Both evaluations happen with substitution depth given by `LEVEL`:

```
>> LEVEL := 1:  eval(a0, a0 + a2);
   LEVEL := 2:  eval(a0, a0 + a2);
   LEVEL := 3:  eval(a0, a0 + a2);
```

```
                    2
a2 + 2, a2 + a4  + 7

   2        2
a4  + 7, a4  + 37

      32, 62
```

Since the default value of `LEVEL` is `100`, `eval` usually has no effect at interactive level:

```
>> delete LEVEL:
   a0, eval(a0), a0 + a2, eval(a0 + a2)

                       32, 32, 62, 62
```

Example 4. This example shows the difference between the evaluation of identifiers and local variables. By default, the value of `LEVEL` is `1` within a procedure, i.e., a global identifier is replaced by its value when evaluated, but there is no further recursive evaluation. This changes when `LEVEL` is assigned a bigger value inside the procedure:

```
>> delete a0, a1, a2, a3:
   a0 := a1 + a2:  a1 := a2 + a3:  a2 := a3^2 - 1:  a3 := 5:
   p := proc()
          save LEVEL;
        begin
          print(a0, eval(a0)):
          LEVEL := 2:
          print(a0, eval(a0)):
        end_proc:

>> p()

                                  2
              a1 + a2, a2 + a3 + a3  - 1

                            2
                 a2 + a3 + a3  - 1, 53
```

In contrast, evaluation of a local variable replaces it by its value, without further evaluation. When `eval` is applied to an object containing a local variable, then the effect is an evaluation of the value of the local variable with substitution depth `LEVEL`:

```
>> q := proc()
          save LEVEL;
          local x;
        begin
          x := a0:
          print(x, eval(x)):
          LEVEL := 2:
          print(x, eval(x)):
        end_proc:
   q()

                                          2
                   a1 + a2, a2 + a3 + a3  - 1

                                     2
                        a1 + a2, a3  + 28
```

The command `x:=a0` assigns the value of the identifier `a0`, namely the unevaluated expression `a1+a2`, to the local variable `x`, and `x` is replaced by this value every time it is evaluated, independent of the value of `LEVEL`:

Example 5. In contrast to lists and sets, evaluation of an array does not evaluate its entries. Thus `eval` has no effect for arrays either. Use `map` to evaluate all entries of an array:

```
>> delete a, b:
   L := [a, b]:  A := array(1..2, L):  a := 1:   b := 2:
   L, A, eval(A), map(A, eval)

                    +-     -+  +-     -+  +-     -+
            [1, 2], | a, b |, | a, b |, | 1, 2 |
                    +-     -+  +-     -+  +-     -+
```

The call `map(A, gamma)` does not evaluate the entries of the array `A` before applying the function `gamma`. Map the function `gamma@eval` to enforce the evaluation:

```
>> map(A, gamma), map(A, gamma@eval)

                +-                    -+  +-     -+
                | gamma(a), gamma(b) |, | 1, 1 |
                +-                    -+  +-     -+
```

Example 6. Similarly, evaluation of a table does not evaluate its entries, and you can use `map` to achieve this. However, this does not affect the indices:

```
>> delete a, b:
   T := table(a = b):  a := 1:  b := 2:
   T, eval(T), map(T, eval)

                    table(    table(    table(
                      a = b ,   a = b ,   a = 2
                    )         )         )
```

If you want a table with evaluated indices as well, create a new table from the evaluated operands of the old table. Using `eval` is necessary here since the operand function `op` does not evaluate the returned operands:

```
>> op(T), table(eval(op(T)))

                                 table(
                          a = b,   1 = 2
                                 )
```

Example 7. `Polynomials` are inert when evaluated, and also `eval` has no effect:

```
>> delete a, x:  p := poly(a*x, [x]):  a := 2:  x := 3:
   p, eval(p), map(p, eval)

         poly(a x, [x]), poly(a x, [x]), poly(a x, [x])
```

Use `mapcoeffs` to evaluate all coefficients:

```
>> mapcoeffs(p, eval)

                          poly(2 x, [x])
```

If you want to substitute a value for the indeterminate `x`, use `evalp`:

```
>> delete x:  evalp(p, x = 3)

                                3 a
```

As you can see, the result of an `evalp` call may contain unevaluated identifiers, and you can evaluate them by an application of `eval`:

```
>> eval(evalp(p, x = 3))

                                 6
```

Example 8. The evaluation of an element of a user-defined `domains` depends on the implementation of the domain. Usually, it is not evaluated further:

```
>> delete a:  T := newDomain("T"):
   e := new(T, a):  a := 1:
   e, eval(e), map(e, eval), val(e)

            new(T, a), new(T, a), new(T, a), new(T, a)
```

If the slot `"evaluate"` exists, the corresponding slot routine is called for a domain element each time it is evaluated. We implement the routine `T::evaluate`, which simply evaluates all internal operands of its argument, for our domain `T`. The unevaluated domain element can still be accessed via `val`:

```
>> T::evaluate := x -> new(T, eval(extop(x))):
   e, eval(e), map(e, eval), val(e)

            new(T, 1), new(T, 1), new(T, 1), new(T, a)
```

`evalassign` – assignment with evaluation of the left hand side

`evalassign(x, value, i)` evaluates `x` with substitution depth `i` and assigns `value` to the result of the evaluation.

Call(s):

- `evalassign(x, value, i)`
- `evalassign(x, value)`

Parameters:

`x`	—	an object that evaluates to a valid left hand side of an assignment
`value`	—	any MuPAD object
`i`	—	a nonnegative integer less than 2^{31}

Return Value: `value`.

Related Functions: `:=`, `_assign`, `assign`, `assignElements`, `delete`, `eval`, `LEVEL`, `level`

Details:

- `evalassign(x, value, i)` evaluates `value`, as usual. Then it evaluates `x` with substitution depth `i`, and finally it assigns the evaluation of `value` to the evaluation of `x`.

The difference between `evalassign` and the assignment operator `:=` is that the latter does not evaluate its left hand side at all.

- As usual, the evaluation of `value` takes place with substitution depth given by `LEVEL`. By default, it is `1` within a procedure.
- See the help pages of `LEVEL` and `level` for the notion of substitution depth and for details about evaluation.
- The third argument is optional. The calls `evalassign(x, value)`, `evalassign(x, value, 0)`, `x := value`, and `_assign(x, value)` are all equivalent.
- The result of the evaluation of `x` must be a valid left hand side for an assignment. See the help page of `:=` for details.
- The second argument is *not* flattened. Hence it may also be a sequence. Cf. example 2.

Example 1. `evalassign` can be used in situations such as the following. Suppose that an identifier `a` has another identifier `b` as its value, and that we want to assign something to this *value* of `a`, not to `a` itself:

```
>> delete a, b: a := b:
   evalassign(a, 100, 1): level(a, 1), a, b

                              b, 100, 100
```

This would not have worked with the assignment operator `:=`, which does not evaluate its left hand side:

```
>> delete a, b: a := b:
   a := 100: level(a, 1), a, b

                              100, 100, b
```

Example 2. The second argument may also be a sequence:

```
>> a := b:
   evalassign(a, (3,5), 1):
   b

                                 3, 5
```

Background:

- The function `level` is used for the evaluation of `x`. Hence `i` may exceed the value of `LEVEL`.
- All special rules for `_assign` apply: see there on further details on indexed assignments, assignments to slots, and the `protect` mechanism.

`evalp` – evaluate a polynomial at a point

`evalp(p, x = v)` evaluates the polynomial `p` in the variable `x` at the point `v`.

Call(s):

- `evalp(p, x = v, ...)`
- `evalp(p, [ x = v, ... ])`
- `evalp(f, <vars,> x = v, ...)`
- `evalp(f, vars, [ x = v, ... ])`

Parameters:

`p`	—	a polynomial of type `DOM_POLY`
`x`	—	an indeterminate
`v`	—	the value for `x`: an element of the coefficient ring of the polynomial
`f`	—	a polynomial expression
`vars`	—	a list of indeterminates of the polynomial: typically, identifiers or indexed identifiers

Return Value: an element of the coefficient ring, or a polynomial, or a polynomial expression, or `FAIL`

Overloadable by: `p, f`

Related Functions: `eval`, `poly`

Details:

- `evalp(p, x = v)` evaluates the polynomial `p` in the variable `x` at the point `v`. An error occurs if `x` is not an indeterminate of `p`. The value `v` may be any object that could also be used as coefficient. The result is an element of the coefficient ring of `p` if `p` is univariate. If `p` is multivariate, the result is a polynomial in the remaining variables.
- If several evaluation points are given, the evaluations take place in succession from left to right. Each evaluation follows the rules above.

- For a polynomial p in the variables x1,x2,..., the syntax p(v1,v2,...) can be used instead of evalp(p,x1=v1,x2=v2,...).
- evalp(f, vars, x = v, ...) first converts the polynomial expression f to a polynomial with the variables given by vars. If no variables are given, they are searched for in f. See poly about details of the conversion. FAIL is returned if f cannot be converted to a polynomial. A successfully converted polynomial is evaluated as above. The result is converted to an expression.
- Horner's rule is used to evaluate the polynomial. The evaluation of variables at the point 0 is most efficient and should take place first. After that, the remaining main variable should be evaluated first.
- The result of evalp is not evaluated further. One may use eval to fully evaluate the result.
- Instead of evalp(p, x1 = v1, x2 = v2, ...) one may also use the equivalent form evalp(p, [x1 = v1, x2 = v2, ...]).
- evalp is a function of the system kernel.

Example 1. evalp is used to evaluate the polynomial expression $x^2 + 2x + 3$ at the point $x = a + 2$. The form of the resulting expression reflects the fact that Horner's rule was used:

```
>> evalp(x^2 + 2*x + 3, x = a + 2)

                    (a + 2) (a + 4) + 3
```

Example 2. evalp is used to evaluate a polynomial in the indeterminates x and y at the point $x = 3$. The result is a polynomial in the remaining indeterminate y:

```
>> p := poly(x^2 + x*y + 2, [x, y]): evalp(p, x = 3)

                    poly(3 y + 11, [y])

>> delete p:
```

Example 3. Polynomials may be called like functions in order to evaluate all variables:

```
>> p := poly(x^2 + x*y, [x, y]): evalp(p, x = 3, y = 2) = p(3, 2)

                          15 = 15

>> delete p:
```

Example 4. If not all variables are replaced by values, the result is a polynomial in the remaining variables:

```
>> evalp(poly(x*y*z + x^2 + y^2 + z^2, [x, y, z]), x = 1, y = 1)

                         2
                 poly(z  + z + 2, [z])
```

Example 5. The result of `evalp` is not evaluated further. We first define a polynomial `p` with coefficient `a` and then change the value of `a`. The change is not reflected by `p`, because polynomials do not evaluate their coefficients implicitly. One must map the function `eval` onto the coefficients in order to enforce evaluation:

```
>> p := poly(x^2 + a*y + 1, [x,y]): a := 2:
   p, mapcoeffs(p, eval)

          2                           2
     poly(x  + a y + 1, [x, y]), poly(x  + 2 y + 1, [x, y])
```

If we use `evalp` to evaluate `p` at the point $x = 1$, the result is not fully evaluated. One must use `eval` to get fully evaluated coefficients:

```
>> r := evalp(p, x = 1):
   r, mapcoeffs(r, eval)

             poly(a y + 2, [y]), poly(2 y + 2, [y])

>> delete p, a, r:
```

Changes:

- The evaluation points may also be given as list.

---

exp – the exponential function

`exp(x)` represents the value of the exponential function at the point x.

Call(s):

- `exp(x)`

Parameters:

`x` — an arithmetical expression or a floating point interval

Return Value: an arithmetical expression or a floating point interval

Overloadable by: `x`

Side Effects: When called with a floating point argument, the function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Related Functions: `ln`, `log`

Details:

- The exponential function is defined for all complex arguments.
- For most exact arguments, an unevaluated function call is returned subject to some simplifications:
  - Calls of the form $\exp(q\,\pi\,i)$ with integer or rational q are rewritten such that q lies in the interval $[0, 2)$. Explicit results are returned if the denominator of q is 1, 2, 3, 4, 5, 6, 8, 10, or 12.
  - Further, the following special values are implemented:
    - $\exp(0) = 1$,
    - $\exp(\texttt{infinity}) = \texttt{infinity}$,
    - $\exp(-\texttt{infinity}) = 0$.
  - A call of the form $\exp(c \ln(y))$ with an unevaluated $\ln(y)$ and a constant c (i.e., of type `Type::Constant`) yields the result y^c.
  - The call $\exp(f(y))$ yields the result $y/f(y)$, if f is `lambertV` or `lambertW`.
- Floating point results are computed, when the argument is a floating point number.

  NOTE

  Numerical overflow/underflow may happen, when the absolute value of the real part of a floating point argument x is large. A protection against underflow is implemented: if $\texttt{Re}(x) < -10^6$, then `exp(x)` may return the truncated result `0.0`. Cf. example 2.
- For arguments of type `DOM_INTERVAL`, the return value is another interval containing the image set of the exponential function over the input interval. See example 4.
- The protected identifier `E` is an alias for `exp(1)`.

Example 1. We demonstrate some calls with exact and symbolic input data:

```
>> exp(1), exp(2), exp(-3), exp(1/4), exp(1 + I), exp(x^2)

                                                   2
     exp(1), exp(2), exp(-3), exp(1/4), exp(1 + I), exp(x )
```

Floating point values are computed for floating point arguments:

```
>> exp(1.23), exp(4.5 + 6.7*I), exp(1.0/10^20), exp(123456.7)

 3.421229536, 82.31014791 + 36.44342846 I, 1.0,

    3.660698702e53616
```

Some special symbolic simplifications are implemented:

```
>> exp(I*PI), exp(x - 22*PI*I), exp(3 + I*PI)

                      -1, exp(x), -exp(3)

>> exp(ln(-2)), exp(ln(x)*PI), exp(lambertW(5))

                          PI       5
                     -2, x  , -----------
                              lambertW(5)
```

Example 2. The truncated result `0.0` may be returned for floating point arguments with negative real parts. This prevents numerical underflow:

```
>> exp(-5.81*10^6), exp(-5.82*10^6)

                   1.148529374e-2523251, 0.0

>> exp(-5.81*10^6 + 10^10*I), exp(-5.82*10^6 + 10^10*I)

       1.002803534e-2523251 - 5.599149896e-2523252 I, 0.0
```

No such protection is implemented for numerical overflow:

```
>> exp(5.81*10^6)

                      8.706786458e2523250

>> exp(5.82*10^6)

 Error: Overflow/underflow in arithmetical operation;
 during evaluation of 'exp::float'
```

Example 3. System functions such as `limit`, `series`, `expand`, `combine` etc. handle expressions involving `exp`:

```
>> limit(x*exp(-x), x = infinity), series(exp(x/(x + 1)), x = 0)

                         2    3    4       5
                        x    x    x    19 x        6
             0, 1 + x - -- + -- + -- - ----- + O(x )
                         2    6   24    120

>> expand(exp(x + y + (sqrt(2) + 5)*PI*I))

                                              1/2
                 - exp(x) exp(y) exp(I PI 2   )

>> combine(%, exp)

                                           1/2
                    - exp(x + y + I PI 2   )
```

Example 4. `exp` transforms intervals (of type `DOM_INTERVAL`) to intervals:

```
>> exp(-1 ... 1)

                   0.3678794411 ... 2.718281829
```

Note that MuPAD's floating point numbers cannot be arbitrarily large. In the context of floating point intervals, all values larger than a machine-dependent constant are regarded as "infinite":

```
>> exp(1 ... 1e1000)

                      2.718281828 ... RD_INF
```

Finally, we would like to mention that you can also use `exp` on disjunct unions of intervals:

```
>> exp((1 ... PI) union (10 ... 20))

  2.718281828 ... 23.14069264 union 22026.46579 ... 485165195.5
```

Changes:

- Arguments of type `DOM_INTERVAL` are now handled.
- `exp(1)^x` is no longer simplified to `exp(x)` automatically.

---

expand – expand an expression

`expand(f)` expands the arithmetical expression `f`.

Call(s):

- `expand(f)`
- `expand(f, g1, g2, ...)`

Parameters:

`f, g1, g2, ...` — arithmetical expressions

Return Value: an arithmetical expression.

Overloadable by: `f`

Side Effects: `expand` is sensitive to properties of identifiers set via `assume`.

Further Documentation: Chapter "Manipulating Expressions" of the Tutorial.

Related Functions: `collect`, `combine`, `denom`, `factor`, `normal`, `numer`, `partfrac`, `rationalize`, `rectform`, `rewrite`, `simplify`

Details:

- The most important use of `expand` is the application of the distributivity law to rewrite products of sums as sums of products. In this respect, `expand` is the inverse function of `factor`.

  Powers of sums with positive integer exponents are expanded as well, but powers of sums with negative integer exponents are not expanded; see example 2.

  The numerator of a fraction is expanded, and then the fraction is rewritten as a sum of fractions with simpler numerators; see example 1. In a certain sense, this is the inverse functionality of `normal`. Use `partfrac` for a more powerful way to rewrite a fraction as a sum of simpler fractions.

- `expand(f)` also applies the following rewriting rules to powers occurring as subexpressions in `f`:

  - $x^{a+b} = x^a x^b$
  - $(xy)^b = x^b y^b$
  - $(x^a)^b = x^{ab}$

  The last two rules are only valid under certain additional restrictions, e.g., when `b` is an integer. Except for the third rule, this behavior of `expand` is the inverse functionality of `combine`. See example 3.

- `expand` works recursively on the subexpressions of an expression `f`. If `f` is of one of the container types `array`, `list`, `set`, or `table`, `expand` only returns `f` and does not map on the entries. If you want to expand all entries of one of the containers please use `map`. See example 4.

- If optional arguments `g1, g2, ...` are present, then any subexpression of `f` that is equal to one of these additional arguments is not expanded; see example 5. See section "Background" for a description how this works.

- Properties of identifiers are taken into account (see `assume`). Identifiers without any properties are assumed to be complex. See example 6.

- `expand` also handles various types of special mathematical functions. It rewrites a single call of a special function with a complicated argument as a sum or a product of several calls of the same function or related functions with simpler arguments. In this respect, `expand` is the inverse function of `combine`.

  In particular, `expand` implements the functional equations of the exponential function and the logarithm, the gamma function and the polygamma function, and the addition theorems for the trigonometric functions and the hyperbolic functions. See example 7.

- `expand` is a function of the system kernel.

Example 1. `expand` expands products of sums by multiplying out:

```
>> expand((x + 1)*(y + z)^2)

                              2    2      2      2
             2 y z + 2 x y z + y  + z  + x y  + x z
```

After expansion of the numerator, a fraction is rewritten as a sum of fractions:

```
>> expand((x + 1)^2*y/(y + z)^2)

                                          2
                    y        2 x y       x  y
                 -------- + -------- + --------
                        2          2          2
                 (y + z)    (y + z)    (y + z)
```

Example 2. Powers of sums with positive integer exponents are expanded:

```
>> expand((x + y)^2)

                                 2    2
                        2 x y + x  + y
```

Powers of sums with negative integer exponents are regarded as denominators of fractions and are not expanded:

```
>> expand((x + y)^(-2))

                                        1
                                    --------
                                           2
                                    (x + y)
```

Example 3. A power with a sum in the exponent is rewritten as a product of powers:

```
>> expand(x^(y + z + 2))

                                     2  y  z
                                    x  x  x
```

If one of the additive terms in the exponent is negative, the power is expanded into a fraction of powers:

```
>> expand((x + y)^(z - 2))

                                           z
                                    (x + y)
                                    --------
                                           2
                                    (x + y)
```

Example 4. `expand` works in a recursive fashion. In the following example, the power $(x+y)^{z+2}$ is first expanded into a product of two powers. Then the power $(x+y)^2$ is expanded into a sum. Finally, the product of the latter sum and the remaining power $(x+y)^z$ is multiplied out:

```
>> expand((x + y)^(z + 2))

                         z    2        z    2        z
           2 x y (x + y)  + x  (x + y)  + y  (x + y)
```

Here is another example:

```
>> expand(2^((x + y)^2))

                                   2   2
                                  x   y   2 x y
                                 2   2   2
```

`expand` does not map on the entries of a container type:

```
>> expand([(a + b)^2, c]), expand({(a + b)^2, c})

                             2                  2
                   [(a + b) , c], {c, (a + b) }
```

Use `map` in order to expand all entries of a container:

```
>> map([(a + b)^2, c], expand), map({(a + b)^2, c}, expand)

                        2    2                    2    2
            [2 a b + a  + b , c], {c, 2 a b + a  + b }
```

Example 5. If additional arguments are provided, `expand` performs only a partial expansion. These additional expressions, such as `x + 1` in the following example, are not expanded:

```
>> expand((x + 1)*(y + z))

                        y + z + x y + x z

>> expand((x + 1)*(y + z), x + 1)

                      y (x + 1) + z (x + 1)
```

Example 6. The following expansions are not valid for all values `a`, `b` from the complex plane. Therefore no expansion is done:

```
>> expand(ln(a^2)), expand(ln(a*b))

                               2
                          ln(a ), ln(a b)
```

The expansions are valid under the assumption that `a` is a positive real number:

```
>> assume(a > 0):  expand(ln(a^2)), expand(ln(a*b))

                     2 ln(a), ln(a) + ln(b)

>> unassume(a):
```

Example 7. The addition theorems of trigonometry are implemented by `"expand"`-slots of the trigonometric functions `sin` and `cos`:

```
>> expand(sin(a + b)), expand(sin(2*a))

         cos(a) sin(b) + cos(b) sin(a), 2 cos(a) sin(a)
```

The same is true for the hyperbolic functions `sinh` and `cosh`:

```
>> expand(cosh(a + b)), expand(cosh(2*a))

                                                 2
        cosh(a) cosh(b) + sinh(a) sinh(b), 2 cosh(a)  - 1
```

The exponential function with a sum as argument is expanded via `exp::expand`:

```
>> expand(exp(a + b))

                          exp(a) exp(b)
```

Here are some more expansion examples for the functions `sum`, `fact`, `abs`, `coth`, `sign`, `binomial`, `beta`, `gamma`, `log`, `cot`, `tan`, `exp` and `psi`:

```
>> sum(x + exp(x),x);  expand(%)

                       sum(x + exp(x), x)

                        2
                       x    x     exp(x)
                       -- - - + ----------
                        2   2   exp(1) - 1

>> fact(x + 1);  expand(%)

                           fact(x + 1)

                         fact(x) (x + 1)

>> abs(a*b);  expand(%)

                            abs(a b)

                          abs(a) abs(b)

>> coth(a + b);  expand(%)

                           coth(a + b)

          cosh(a) cosh(b)
 --------------------------------- +
 cosh(a) sinh(b) + cosh(b) sinh(a)

             sinh(a) sinh(b)
    ---------------------------------
    cosh(a) sinh(b) + cosh(b) sinh(a)
```

```
>> coth(a*b);  expand(%)

                           coth(a b)

                           cosh(a b)
                           ---------
                           sinh(a b)

>> sign(a*b);  expand(%)

                           sign(a b)

                        sign(a) sign(b)

>> tan(a);  expand(%)

                            tan(a)

                            sin(a)
                            ------
                            cos(a)

>> binomial(n, m);  expand(%)

                        binomial(n, m)

                           n gamma(n)
                 -------------------------------
                 m gamma(m) (n - m) gamma(n - m)

>> beta(n, m);  expand(%)

                          beta(m, n)

                       gamma(m) gamma(n)
                       -----------------
                          gamma(m + n)

>> gamma(x+1);  expand(%)

                         gamma(x + 1)

                          x gamma(x)

>> log(10, x); expand(%)

                          log(10, x)

                             ln(x)
                            ------
                            ln(10)
```

```
>> cot(x);  expand(%)

                              cot(x)

                              cos(x)
                              ------
                              sin(x)

>> exp(x + y);  expand(%)

                            exp(x + y)

                          exp(x) exp(y)

>> psi(x + 2);  expand(%)

                            psi(x + 2)

                                1     1
                       psi(x) + - + -----
                                x   x + 1
```

Example 8. This example illustrates how to extend the functionality of `expand` to user-defined mathematical functions. As an example, we consider the sine function. (Of course, the system function `sin` already has an `"expand"` slot; see example 7.)

We first embed our function into a function environment, which we call `Sin`, in order not to overwrite the system function `sin`. Then we implement the addition theorem $\sin(x+y) = \sin(x)\cos(y) + \sin(y)\cos(x)$ in the `"expand"` slot of the function environment, i.e., the slot routine `Sin::expand`:

```
>> Sin := funcenv(Sin):
   Sin::expand := proc(u) // compute expand(Sin(u))
     local x, y;
   begin
     // recursively expand the argument u
     u := expand(u);

     if type(u) = "_plus" then // u is a sum

       x := op(u, 1); // the first term
       y := u - x;    // the remaining terms

       // apply the addition theorem and
       // expand the result again
       expand(Sin(x)*cos(y) + cos(x)*Sin(y))
```

```
    else
      Sin(u)
    end_if
  end_proc:
```

Now, if expand encounters a subexpression of the form Sin(u), it calls Sin::expand(u) to expand Sin(u). The following command first expands the argument a*(b+c) via the recursive call in Sin::expand, then applies the addition theorem, and finally expand itself expands the product of the result with z:

```
>> expand(z*Sin(a*(b + c)))

           z Sin(a b) cos(a c) + z Sin(a c) cos(a b)
```

The expansion after the application of the addition theorem in Sin::expand is necessary to handle the case when u is a sum with more than two terms: then y is again a sum, and cos(y) and Sin(y) are expanded recursively:

```
>> expand(Sin(a + b + c))

 Sin(a) cos(b) cos(c) + Sin(b) cos(a) cos(c) +

    Sin(c) cos(a) cos(b) - Sin(a) sin(b) sin(c)
```

Background:

- With optional arguments g1, g2, ..., the expansion of certain subexpressions of f can be prevented. This works as follows: every occurrence of g1, g2, ... in f is replaced by an auxiliary variable before the expansion, and afterwards the auxiliary variables are replaced by the original subexpressions.
- Users can extend the functionality of expand to their own special mathematical functions via overloading. To this end, embed your function into a function environment g and implement the behavior of expand for this function in the "expand" slot of the function environment.

  Whenever expand encounters a subexpression of the form g(u,..), it issues the call g::expand(u,..) to the slot routine to expand the subexpression, passing the not yet expanded arguments u,.. of g as arguments. The result of this call is not expanded any further by expand. See example 8 above.
- Similarly, an "expand" slot can be defined for a user-defined library domain T. Whenever expand encounters a subexpression d of domain type T, it issues the call T::expand(d) to the slot routine to expand d. The result of this call is not expanded any further by expand. If T has no "expand" slot, then d remains unchanged.

`export`, `unexport` – export library functions or undo the export

`export(L, f)` exports the public function `L::f` of the library `L`, such that it can be accessed as `f`, without the prefix `L`.

`export(L)` exports all public functions of the library `L`.

`unexport(L, f)` undoes the export of the public function `L::f` of the library `L`, such that it is no longer available as `f`.

`unexport(L)` undoes the export of all previously exported public functions of the library `L`.

Call(s):

- `export(L, f1, f2, ...)`
- `export(L)`
- `unexport(L, f1, f2, ...)`
- `unexport(L)`

Parameters:

`L`	—	the library: a domain
`f1, f2, ...`	—	public functions of `L`: identifiers

Return Value: the void object `null()` of type `DOM_NULL`.

Side Effects: When a function is exported, it is assigned to the corresponding global identifier. When it is unexported, the corresponding identifier is deleted.

Further Documentation: Chapter "The MuPAD libraries" of the Tutorial.

Related Functions: `:=`, `delete`, `info`, `loadmod`, `loadproc`, `package`, `unloadmod`

Details:

- A library contains *public* functions which may be called by the user. The collection of these functions forms the *interface* of the library. (There may be other, private, functions, too, which are not intended to be called by the user directly, and are not documented.) The standard way of accessing the public function `f` from the library `L` is via `L::f`. When the function `f` is *exported*, it can be accessed more briefly as `f`. Technically, exporting means that the global identifier `f` is assigned the value `L::f`.

- On the other hand, unexporting the library function `f` means that the value of the global identifier `f` is deleted. Afterwards, the library function is available only as `L::f`.
- `export(L, f1, f2, ...)` exports the given functions `f1, f2, ...` of `L`. However, if one of the identifiers already has a value, the corresponding function is not exported. A warning is printed instead. An error is returned if one of the identifiers is not the name of a public library function.
- `export(L)` exports all public functions of `L`.
- A function that is already exported will not be exported twice. A warning is printed instead.
- `unexport(L, f1, f2, ...)` unexports all given functions of `L`. Note that `unexport` does not evaluate the identifiers. Thus, it is not necessary to use `hold` to protect them from being evaluated.
- `unexport(L)` unexports all public functions of the library `L`.
- `export` and `unexport` evaluate their first argument `L`, but they do not evaluate the remaining arguments `f1, f2, ...`, if any.
- The function `info` displays the interface functions and the exported functions of a library.
- Some libraries have functions that are always exported. These functions cannot be unexported. The function `append` from the library `listlib` is such an example.

  Most functions of the standard library `stdlib` are exported automatically.

Example 1. We export the public function `invphi` of the library `numlib` and then undo the export:

```
>> numlib::invphi(4!)

            [35, 39, 45, 52, 56, 70, 72, 78, 84, 90]

>> export(numlib, invphi):

>> invphi(4!)

            [35, 39, 45, 52, 56, 70, 72, 78, 84, 90]

>> unexport(numlib, invphi):

>> invphi(4!)

                              invphi(24)
```

We export and unexport all public functions of the library `numlib`:

```
>> export(numlib):
   invphi(100)
```

```
 Warning: 'contfrac' already has a value, not exported.

                     [101, 125, 202, 250]
```

As you can see `export` issued a warning because `contfrac` already has a value. Here, the reason in the existence of a global function `contfrac` which makes use of `numlib::contfrac` for numerical arguments.

```
>> unexport(numlib):
   invphi(100)
```

```
                          invphi(100)
```

Example 2. `export` issues a warning if a function cannot be exported since the corresponding identifier already has a value:

```
>> invphi := 17:
   export(numlib, invphi)
```

```
 Warning: 'invphi' already has a value, not exported.
```

A function will not be exported twice, and `export` issues a corresponding message if you try:

```
>> delete invphi:
   export(numlib, invphi):
   export(numlib, invphi):
   unexport(numlib, invphi):
```

```
 Info: 'numlib::invphi' already is exported.
```

Background:

- The names of the public functions of a library `L` are stored in the set `L::interface`. This set is used by the function `info` and for exporting.
- The names of functions exported from a library `L` are stored in the set `L::exported`.

---

expose – display the source code of a procedure or the entries of a domain

`expose(f)` displays the source code of the MuPAD procedure `f` or the entries of the domain `f`.

Call(s):

- `expose(f)`

Parameters:

`f` — any object; typically, a procedure, a function environment, or a domain

Return Value:

- If `f` is a procedure, `expose` returns the complete source code of `f`, of type `stdlib::Exposed` (see "Background" below).
- If `f` is a function environment, the result of applying `expose` to the first operand is returned.
- If `f` is a domain, `expose` returns a symbolic call to `newDomain`; see below for details.
- In all other cases, `expose` returns `f` if it is not overloaded.

Side Effects: The formatting of the output of `expose` is sensitive to the environment variable `TEXTWIDTH`.

Overloadable by: `f`

Related Functions: `print`

Details:

- Usually, procedures and domains are printed in abbreviated form. `expose` serves to display the complete source code of a procedure and all entries of a domain, respectively.
- If `f` is a domain, then `expose` returns a symbolic `newDomain` call. The arguments of the call are equations of the form `index = value`, where `value` equals the value of `f::index`. `expose` is not recursively applied to `f::index`; hence, the source code of domain methods is not displayed.
- Although `expose` returns a syntactically valid MuPAD object, this return value is intended for screen output only, and further processing of it is deprecated.

Example 1. Using `expose`, you can inspect the source code of procedures of the MuPAD library:

```
>> sin
```

```
                              sin
```

```
>> expose(%)
```

```
 proc(x)
   name sin;
   local f, y;
   option noDebug;
 begin
   if args(0) = 0 then
     error("no arguments given")
   else
   ...
 end_proc
```

Example 2. On the other hand, you cannot look at the source code of kernel functions:

```
>> expose(_plus)
```

```
                 builtin(817, NIL, "_plus", NIL)
```

Example 3. When applied to a domain, `expose` shows the entries of that domain:

```
>> expose(DOM_INT)
```

```
       domain DOM_INT
         D := 0;
         new := proc new() ... end;
         new_extelement := proc new_extelement(d) ... end;
         phi := phi;
         coerce := proc DOM_INT::coerce(x) ... end;
       end_domain
```

Example 4. Applying `expose` to other objects is legal but generally useless:

```
>> expose(3)
```

```
                               3
```

Background:

- In addition to the usual `overloading` mechanism for domain elements, a domain method overloading `expose` must handle the following case: it will be called with zero arguments when the domain itself is to be exposed.
- If `f` is a procedure, then `expose` returns an object of the domain `stdlib::Exposed`. The only purpose of this domain is its `"print"` method; manipulating its elements should never be necessary. Therefore it remains undocumented.

`expr` – convert into an element of a basic domain

`expr(object)` converts `object` into an element of a basic domain, such that all sub-objects are elements of basic domains as well.

Call(s):

- `expr(object)`

Parameters:

`object` — an arbitrary object

Return Value: an element of a basic domain.

Overloadable by: `object`

Related Functions: `coerce`, `domtype`, `eval`, `testtype`, `type`

Details:

- `expr` is a type conversion function, for converting an element of a more complex library domain, such as a polynomial or a matrix, into an element of a basic kernel domain.

  `expr` proceeds recursively, such that all sub-objects of the returned object are elements of basic domains as well. See example 2.
- The two special objects `infinity` and `complexInfinity` are translated into identifiers with the same name by `expr`. Evaluating these identifiers yields the original objects. See example 1.
- If `object` already belongs to a basic domain other than `DOM_POLY`, then `expr` is only applied recursively to the operands of `object`, if any.
- If `object` is a polynomial of domain type `DOM_POLY`, then `expr` is applied recursively to the coefficients of `object`, and afterwards the result is converted into an identifier, a number, or an expression. See example 1.

- If `object` belongs to a library domain `T` with an `"expr"` slot, then the corresponding slot routine `T::expr` is called with `object` as argument, and the result is returned.

  This can be used to extend the functionality of `expr` to elements of user-defined domains. If the slot routine is unable to perform the conversion, it must return `FAIL`. See example 6.

  If the domain `T` does not have an `"expr"` slot, then `expr` returns `FAIL`.

- The result of `expr` is not evaluated further. Use `eval` to evaluate it. See example 4.

Example 1. `expr` converts a polynomial into an expression, an identifier, or a number:

```
>> expr(poly(x^2 + y, [x])), expr(poly(x)), expr(poly(2, [x]));
   map(%, domtype)

                              2
                         y + x , x, 2

                  DOM_EXPR, DOM_IDENT, DOM_INT
```

The objects `infinity` and `complexInfinity` are translated into identifiers with the same names:

```
>> expr(infinity), expr(complexInfinity);
   map(%, domtype)

                   infinity, complexInfinity

                      DOM_IDENT, DOM_IDENT
```

If these identifiers are evaluated with `eval` the results are the original objects of the types `stdlib::Infinity` and `stdlib::CInfinity`:

```
>> expr(infinity), expr(complexInfinity);
   map(eval(%), domtype)

                   infinity, complexInfinity

             stdlib::Infinity, stdlib::CInfinity
```

Example 2. This example shows that `expr` works recursively on expressions. All subexpressions which are domain elements are converted into expressions. In earlier versions of MuPAD (up to version 1.4.2) the result would have been `x + (1 mod 7)`. The construction with `hold(_plus)(..)` is necessary since `x + i(1)` would evaluate to `FAIL`:

```
>> i := Dom::IntegerMod(7):
   hold(_plus)(x, i(1));  expr(%)

                              x + (1 mod 7)

                                  x + 1
```

Example 3. The function `series` returns an element of the domain `Series::Puiseux`, which is not a basic domain:

```
>> s := series(sin(x), x);
   domtype(s)

                             3     5
                            x     x        7
                        x - -- + --- + O(x )
                            6    120

                           Series::Puiseux
```

Use `expr` to convert the result into an element of domain type `DOM_EXPR`:

```
>> e := expr(s); domtype(e)

                                 3     5
                                x     x
                            x - -- + ---
                                6    120

                              DOM_EXPR
```

Note that the information about the order term is lost after the conversion.

Example 4. `expr` does not evaluate its result. In this example the polynomial `p` has a parameter `a` and the global variable `a` has a value. `expr` applied on the polynomial `p` returns an expression containing `a`. If you want to insert the value of `a` use the function `eval`:

```
>> p := poly(a*x, [x]):  a := 2:  expr(p);  eval(%)

                                 a x

                                 2 x
```

Example 5. `A` is an element of type `Dom::Matrix(Dom::Integer)`:

```
>> A := Dom::Matrix(Dom::Integer)([[1, 2], [3, 2]]);
   domtype(A)

                              +-       -+
                              |  1, 2   |
                              |         |
                              |  3, 2   |
                              +-       -+

                         Dom::Matrix(Dom::Integer)
```

In this case, `expr` converts `A` into an element of type `DOM_ARRAY`:

```
>> a := expr(A); domtype(a)

                              +-       -+
                              |  1, 2   |
                              |         |
                              |  3, 2   |
                              +-       -+

                               DOM_ARRAY
```

However, it is not guaranteed that the result is of type `DOM_ARRAY` in future versions of MuPAD as well. For example, the internal representation of matrices might change in the future. Use `coerce` to request the conversion into a particular data type:

```
>> coerce(A, DOM_ARRAY)

                              +-       -+
                              |  1, 2   |
                              |         |
                              |  3, 2   |
                              +-       -+
```

A nested `list` is an alternative representation for a matrix:

```
>> coerce(A, DOM_LIST)

                            [[1, 2], [3, 2]]
```

Example 6. If a sub-object belongs to a domain without an `"expr"` slot, then `expr` returns `FAIL`:

```
>> T := newDomain("T"):
   d := new(T, 1, 2);
   expr(d)
```

new(T, 1, 2)

FAIL

You can extend the functionality of `expr` to your own domains. We demonstrate this for the domain `T` by implementing an `"expr"` slot, which returns a list with the internal operands of its argument:

```
>> T::expr := x -> [extop(x)]:
```

If now `expr` encounters a sub-object of type `T` during the recursive process, it calls the slot routine `T::expr` with the sub-object as argument:

```
>> expr(d), expr([d, 3])
```

[1, 2], [[1, 2], 3]

`expr2text` – convert objects into character strings

`expr2text(object)` converts `object` into a character string.

Call(s):

- `expr2text(object)`

Parameters:

`object` — any MuPAD object

Return Value: a `string`.

Overloadable by: `object`

Related Functions: `coerce`, `fprint`, `int2text`, `tbl2text`, `text2expr`, `text2int`, `text2list`, `text2tbl`, `print`

Details:

- `expr2text(object)` converts `object` into a character string. The result usually corresponds to the screen output of `object` when `PRETTYPRINT` is set to `FALSE`.

- If the function is called without arguments, then an empty character string is created. If more than one argument is given, the arguments are interpreted as an expression sequence and are converted into a single character string.
- Like most other MuPAD function, `expr2text` evaluates its arguments before the conversion.
- If strings occur in `object`, they will be quoted in the result.
- `expr2text` is a function of the system kernel.

Example 1. Expressions are converted into character strings:

```
>> expr2text(a + b)

                         "a + b"
```

`expr2text` quotes strings. Note that the quotation marks are preceded by a backslash when they are printed on the screen:

```
>> expr2text(["text", 2])

                      "[\"text\", 2]"
```

Example 2. If more than one argument is given, the arguments are treated as a single expression sequence:

```
>> expr2text(a, b, c)

                        "a, b, c"
```

If no argument is given, an empty string is generated:

```
>> expr2text()

                           ""
```

Example 3. `expr2text` evaluates its arguments:

```
>> a := b: c := d: expr2text(a, c)

                         "b, d"
```

Use `hold` to prevent evaluation:

```
>> expr2text(hold(a, c));
   delete a, c:
```

"a, c"

Here is another example:

```
>> expr2text((a := b; c := d));
   delete a, c:
```

"d"

```
>> e := expr2text(hold((a := b; c := d)))
```

"(a := b; \nc := d)"

The last string contains a newline character '`\n`'. Use `print` with option `Unquoted` to expand this into a new line:

```
>> print(Unquoted, e):
```

```
(a := b;
c := d)
```

Example 4. `expr2text` is overloadable. It uses a default output for elements of a domain if the domain has neither a `"print"` slot nor an `"expr2text"` slot:

```
>> T := newDomain("T"): e := new(T, 1):
   e;
   print(e):
   expr2text(e)
```

new(T, 1)

new(T, 1)

"new(T, 1)"

If a `"print"` slot exists, it will be called by `expr2text` to generate the output:

```
>> T::print := proc(x) begin
     _concat("foo: ", expr2text(extop(x)))
   end_proc:
   e;
   print(e):
   expr2text(e)
```

```
foo: 1

foo: 1

"foo: 1"
```

If you want `expr2text` to generate an output differing from the usual output generated by `print`, you can supply an `"expr2text"` method:

```
>> T::expr2text := proc(x) begin
     _concat("bar: ", expr2text(extop(x)))
   end_proc:
   e;
   print(e):
   expr2text(e)
```

```
foo: 1

foo: 1

"bar: 1"
```

Background:

- When processing a domain element `e`, `expr2text` first tries to call the `"expr2text"` method of the corresponding domain `T`. If it exists, `T::expr2text(e)` is called and the result is returned. If no `"expr2text"` method exists, `expr2text` tries to call the `"print"` method in the same way. If no `"print"` method exists either, `expr2text` will generate a default output. Cf. example 4.

  An `"expr2text"` method or a `"print"` method may return an arbitrary MuPAD object, which will be processed recursively by `expr2text`.

  NOTE The returned object must not contain the domain element `e` as a sub-object. Otherwise, the MuPAD kernel runs into infinite recursion and emits an error message.

- For expressions, the result returned by `expr2text` always coincides with the output produced by `print`. If the 0th operand of the expression is a function environment, the result of `expr2text` is computed by the second operand of the function environment.

`external` – create a module function environment

`external("mstring", "fstring")` returns the function environment of the module function `mstring::fstring`.

Call(s):

- `external("mstring", "fstring")`

Parameters:

`"mstring"` — the name of a module: a character string
`"fstring"` — the name of a module function: a character string

Return Value: a function environment of type `DOM_FUNC_ENV`.

Related Functions: `loadmod`, `module::new`, `unloadmod`

Details:

- `external("mstring", "fstring")` creates and returns the function environment of the module function `mstring::fstring`.
- There may be a file `mstring.mdg` containing MuPAD objects that are loaded and bound to the module function environment. If an error occurs while loading these objects, a warning is displayed. MuPAD keeps trying to load them at each subsequent call of module functions affected by it.
- Using `external`, a module function can be accessed without loading the module explicitly and without creating the module domain. If such a module function is executed, its machine code is loaded automatically if necessary.
- Some module functions may only work correctly if their module domain was created before. Such modules must be loaded with `loadmod` before any of their module functions are executed. Refer to the documentation of the corresponding module.
- `external` is a function of the system kernel.

Example 1. Module function environments can be stored in local or global variables. They can be used to execute module functions without loading the module explicitly:

```
>> where := external("stdmod", "which"): where("stdmod")

            "/usr/local/mupad/linux/modules/stdmod.mdm"

>> delete where:
```

Background:

- The kernel functions `external`, `loadmod` and `unloadmod` provide basic features for accessing modules. Extended features are available with the `module` library.

`extnops` – the number of operands of a domain element

`extnops(object)` returns the number of operands of the object's internal representation.

Call(s):

- `extnops(object)`

Parameters:

`object` — an arbitrary MuPAD object

Return Value: a nonnegative integer.

Related Functions: `DOM_DOMAIN`, `extop`, `extsubsop`, `new`, `nops`, `op`, `subsop`

Details:

- For objects of a basic data type such as expressions, sets, lists, tables, arrays etc., `extnops` yields the same result as the function `nops`. The only difference to the function `nops` is that `extnops` cannot be overloaded by domains implemented in the MuPAD language.
- Internally, a domain element may consist of an arbitrary number of data objects; `extnops` returns the actual number of *internal* operands. Since every domain should provide interface methods, `extnops` should only be used from inside these methods. "From the outside", the function `nops` should be used.
- `extnops` is a function of the system kernel.

Example 1. `extnops` returns the number of entries of a domain element:

```
>> d := newDomain("demo"): e := new(d, 1, 2, 3, 4): extnops(e)

                                   4

>> delete d, e:
```

Example 2. For kernel domains, `extnops` is equivalent to `nops`:

```
>> extnops([1, 2, 3, 4]), nops([1, 2, 3, 4])

                              4, 4
```

Example 3. We define a domain of lists. Its internal representation is a single object (a list of kernel type `DOM_LIST`):

```
>> myList := newDomain("lists"):
   myList::new := proc(l : DOM_LIST) begin new(myList, l) end_proc:
```

We want the functionality of `nops` for this domain to be the same as for the kernel type `DOM_LIST`. To achieve this, we overload the function `nops`. The internal list is accessed via `extop(l, 1)`:

```
>> myList::nops := l -> nops(extop(l, 1)):
```

We create an element of this domain:

```
>> mylist := myList([1, 2, 3])

                      new(lists, [1, 2, 3])
```

Since `nops` was overloaded, `extnops` provides the only way of determining the number of operands of the internal representation of `mylist`. In contrast to `nops`, `extnops` always returns 1, because the internal representation consists of exactly one list:

```
>> nops(mylist), extnops(mylist)

                              3, 1

>> delete myList, mylist:
```

`extop` – the operands of a domain element

`extop(object)` returns all operands of the domain element `object`.

`extop(object, i)` returns the `i`-th operand.

`extop(object, i..j)` returns the `i`-th to `j`-th operand.

Call(s):

- `extop(object)`
- `extop(object, i)`
- `extop(object, i..j)`

Parameters:

`object`	—	an arbitrary MuPAD object
`i, j`	—	nonnegative integers

Return Value: a sequence of operands or the specified operand. `FAIL` is returned if no corresponding operand exists.

Related Functions: `DOM_DOMAIN`, `extnops`, `extsubsop`, `new`, `nops`, `op`, `subsop`

Details:

- For objects of a basic data type such as expressions, sets, lists, tables, arrays etc., `extop` yields the same operands as the function `op`. See the corresponding documentation for details on operands. The main difference to the function `op` is that `extop` cannot be overloaded. Therefore, it guarantees direct access to the operands of the *internal representation* of elements of a library domain. Typically, `extop` is used in the implementation of the `"op"` method of a library domain that overloads the system's `op` function.
- A domain element consists of a reference to the corresponding domain and a sequence of values representing its contents. The function `extop` allows access to the domain and the operands of this internal data sequence.
- `extop(object)` returns a sequence of all internal operands except the 0-th one. This call is equivalent to `extop(object, 1..extnops(object))`.
- `extop(object, i)` returns the `i`-th internal operand. In particular, the domain of the object is returned by `extop(object, 0)` if `object` is an element of a library domain. If `object` is an element of a kernel domain, the call `extop(object, 0)` is equivalent to `op(object, 0)`.
- `extop(object, i..j)` returns the `i`-th to `j`-th internal operands of `object` as an expression sequence; `i` and `j` must be nonnegative integers with `i` smaller or equal to `j`. This sequence is equivalent to `extop(object, k) $ k = i..j`.
- `extop` returns `FAIL` if a specified operand does not exist. Cf. example 4.
- The operands of an expression sequence are its elements. Note that such sequences are not flattened by `extop`.
- `extop` is a function of the system kernel.

Example 1. We create a new domain `d` and use the function `new` to create an element of this type. Its internal data representation is the sequence of arguments passed to `new`:

```
>> d := newDomain("demo"): e := new(d, 1, 2, 3): extop(e)

                        1, 2, 3
```

Individual operands can be selected:

```
>> extop(e, 2)

                           2
```

Ranges of operands can be selected:

```
>> extop(e, 1..2)

                         1, 2
```

The 0-th operand of a domain element is its domain:

```
>> extop(e, 0)

                         demo

>> delete d, e:
```

Example 2. First, a new domain `d` is defined via `newDomain`. The `"new"` method serves for creating elements of this type. The internal representation of the domain is a sequence of all arguments of this `"new"` method:

```
>> d := newDomain("d"): d::new := () -> new(dom, args()):
```

The system's `op` function is overloaded by the following `"op"` method of this domain. It is to return the elements of a sorted copy of the internal data sequence. In the implementation of the `"op"` method, the function `extop` is used to access the internal data:

```
>> d::op := proc(x, i = null())
            local internalData;
            begin internalData := extop(x);
                  op(sort([internalData]), i)
            end_proc:
```

Due to this overloading, `op` returns different operands than `extop`:

```
>> e := d(3, 7, 1): op(e); extop(e)

                        1, 3, 7

                        3, 7, 1

>> delete d, e:
```

Example 3. For kernel data types such as sets, lists etc., `extop` always returns the same operands as `op`:

```
>> extop([a, b, c]) = op([a, b, c])

                    (a, b, c) = (a, b, c)
```

Expressions are of kernel data type `DOM_EXPR`, thus `extop(sin(x), 0)` is equivalent to `op(sin(x), 0)`:

```
>> domtype(sin(x)), extop(sin(x), 0) = op(sin(x), 0)

                     DOM_EXPR, sin = sin
```

Expression sequences are not flattened:

```
>> extop((1, 2, 3), 0), extop((1, 2, 3))

                      _exprseq, 1, 2, 3
```

Example 4. Non-existing operands are returned as `FAIL`:

```
>> extop([1, 2], 4),  extop([1, 2], 1..4)

                         FAIL, FAIL
```

`extsubsop` – substitute operands of a domain element

`extsubsop(d, i = new)` returns a copy of the domain element `d` with the `i`-th operand of the internal representation replaced by `new`.

Call(s):

- `extsubsop(d, i1 = new1, i2 = new2, ...)`

Parameters:

`d`	—	arbitrary MuPAD object
`i1, i2, ...`	—	nonnegative integers
`new1, new2, ...`	—	arbitrary MuPAD objects

Return Value: the input object with replaced operands.

Related Functions: `DOM_DOMAIN`, `extnops`, `extop`, `new`, `nops`, `op`, `subs`, `subsex`, `subsop`

Details:

- Internally, a domain element may consist of an arbitrary number of objects. `extsubsop` replaces one or more of these objects, without checking whether the substitution is meaningful.

  The operands of elements of domains of the MuPAD library must meet certain (undocumented) conditions; use `extsubsop` only for your own domains. It is good programming style to use `extsubsop` only inside low-level domain methods.

- `extsubsop` returns a modified copy of the object, but does not change the object itself.
- The numbering of operands is the same as the one used by `extop`.
- If the 0-th operand is to be replaced, the corresponding new value must be a domain of type `DOM_DOMAIN`; `extsubsop` then replaces the domain of `d` by this new domain.
- When trying to replace the `i`-th operand with `i` exceeding the actual number of operands, `extsubsop` first increases the number of operands by appending as many `NIL`'s as necessary and then performs the substitution. Cf. example 3.
- When the `i`-th operand is replaced by an expression sequence of `k` elements, each of these elements becomes an individual operand of the result, indexed from `i` to `i+k-1`. The remaining operands of `d` are shifted to the right accordingly. This new numbering is already in effect for the remaining substitutions in the same call to `extsubsop`. Cf. example 4.
- The void object `null()` becomes an operand of the result when it is substituted into an object.
- After performing the substitution, `extsubsop` does not evaluate the result once more. Cf. example 5.
- In contrast to the function `subsop`, `extsubsop` cannot be overloaded.
- Like `extop` and `extnops`, `extsubsop` can be applied to objects of a kernel domain. In this case `extsubsop` behaves like `subsop`.
- `extsubsop` is a function of the system kernel.

Example 1. We create a domain element and then replace its first operand:

```
>> d := newDomain("1st"): e := new(d, 1, 2, 3): extsubsop(e, 1 = 5)

                        new(1st, 5, 2, 3)
```

This does not change the value of `e`:

```
>> e
```

```
                    new(1st, 1, 2, 3)
```

```
>> delete d, e:
```

Example 2. The domain type of an element can be changed by replacing its 0-th operand:

```
>> d := newDomain("some_domain"): e := new(d, 2):
   extsubsop(e, 0 = Dom::IntegerMod(5))
```

```
                         2 mod 5
```

```
>> delete d, e:
```

Example 3. We substitute the sixth operand of a domain element that has less than six operands. In such cases, an appropriate number of `NIL`'s is inserted:

```
>> d := newDomain("example"): e := new(d, 1, 2, 3, 4):
   extsubsop(e, 6 = 8)
```

```
             new(example, 1, 2, 3, 4, NIL, 8)
```

```
>> delete d, e:
```

Example 4. We substitute the first operand of a domain element `e` by a sequence with three elements. These become the first three operands of the result; the second operand of `e` becomes the fourth operand of the result, and so on. This new numbering is already in effect when the second substitution is carried out:

```
>> d := newDomain("example"): e := new(d, 1, 2, 3, 4):
   extsubsop(e, 1 = (11, 13, 17), 2 = (29, 99))
```

```
           new(example, 11, 29, 99, 17, 2, 3, 4)
```

```
>> delete d, e:
```

Example 5. We define a domain with its own evaluation method. This method prints out its argument such that we can see whether it is called. Then we define an element of our domain.

```
>> d := newDomain("anotherExample"):
   d::evaluate := x -> (print("Argument:", x); x):
   e := new(d, 3)

                    new(anotherExample, 3)
```

We can now watch all evaluations that happen: `extsubsop` evaluates its arguments, performs the desired substitution, but does not evaluate the result of the substitution:

```
>> extsubsop(e, 1 = 0)

              "Argument:", new(anotherExample, 3)

                    new(anotherExample, 0)

>> delete d, e:
```

Example 6. `extsubsop` applied to an object from a kernel type yields the same result as `subsop`:

```
>> extsubsop([1,2,3], 2=4), subsop([1,2,3], 2=4)

                     [1, 4, 3], [1, 4, 3]
```

Changes:

- `extsubsop` now works for kernel data types, too.

`fact` – the factorial function

`fact(n)` represents the factorial $n! = 1 \times 2 \times 3 \times \cdots \times n$ of an integer.

Call(s):

- `fact(n)`
- `n!`

Parameters:

`n` — an arithmetical expression representing a nonnegative integer

Return Value: an arithmetical expression.

Overloadable by: `n`

Related Functions: `beta`, `binomial`, `gamma`, `igamma`, `psi`

Details:

- The short hand call `n!` is equivalent to `fact(n)`.
- If `n` is a nonnegative integer, then an integer is returned. If `n` is a numerical value of some other type, then an error occurs. If `n` is a symbolic expression, then a symbolic call of `fact` is returned.
- Integer arguments must be smaller than 2^{31} on 32-bit systems and smaller than 2^{63} on 64-bit systems, respectively. Larger integers lead to an error.
- The `gamma` function generalizes the factorial function to arbitrary complex arguments. It satisfies `gamma(n+1) = n!` for nonnegative integers `n`. Expressions involving symbolic `fact` calls can be rewritten by `rewrite(expression, gamma)`. Cf. example 3.
- The operator `!` can also be used in prefix notation with an entirely different meaning: `!command` is equivalent to `system("command")`.
- `fact` is a function of the system kernel.

Example 1. Integer numbers are produced if the argument is a nonnegative integer:

```
>> fact(0), fact(5), fact(2^5)

          1, 120, 263130836933693530167218012160000000
```

A symbolic call is returned if the argument is a symbolic expression:

```
>> fact(n), fact(n - sin(x)), fact(3.0*n + I)

           fact(n), fact(n - sin(x)), fact(3.0 n + I)
```

The calls `fact(n)` and `n!` are equivalent:

```
>> 5! = fact(5), (n^2 + 3)!

                                     2
                     120 = 120, fact(n  + 3)
```

A numerical argument produces an error if it is not a positive integer:

```
>> fact(3/2 + I)

 Error: Non-negative integer expected [specfunc::fact];
 during evaluation of 'fact'
```

Example 2. Use `gamma(float(n+1))` rather than `float(fact(n))` for floating point approximations of large factorials. This avoids the costs of computing large integer numbers:

```
>> float(fact(2^13)) = gamma(float(2^13 + 1))

              1.275885799e28503 = 1.275885799e28503
```

Example 3. The functions `expand`, `limit`, `rewrite` and `series` handle expressions involving `fact`:

```
>> expand(fact(n^2 + 4))

                 2    2         2         2         2
          fact(n ) (n  + 1) (n  + 2) (n  + 3) (n  + 4)

>> limit(fact(n)/exp(n), n = infinity)

                             infinity

>> rewrite(fact(2*n^2 + 1)/fact(n - 1), gamma)

                                   2
                          gamma(2 n  + 2)
                          ---------------
                             gamma(n)
```

The Stirling formula is obtained as an asymptotic series:

```
>> series(fact(n), n = infinity, 3)

      1/2   1/2  n  1/2       1/2  n  1/2    /       n     \
     n    PI    n  2        PI    n  2       |      n      |
     ------------------ + -------------- + O| ----------- |
           exp(n)               1/2          |  3/2        |
                          12 n    exp(n)     \ n    exp(n) /
```

`factor` – factor a polynomial into irreducible polynomials

`factor(f)` computes a factorization $f = u \cdot f_1^{e_1} \cdot \ldots \cdot f_r^{e_r}$ of the polynomial f, where u is the content of f, $f_1, \ldots, f_r$ are the distinct primitive irreducible factors of f, and $e_1, \ldots, e_r$ are positive integers.

Call(s):

- `factor(f)`

Parameters:

`f` — a polynomial or an arithmetical expression

Return Value: an object of the domain type `Factored`.

Overloadable by: `f`

Further Documentation: Chapter “Manipulating Expressions” of the Tutorial.

Related Functions: `collect`, `content`, `denom`, `div`, `divide`, `expand`, `Factored`, `gcd`, `icontent`, `ifactor`, `igcd`, `ilcm`, `indets`, `irreducible`, `isprime`, `lcm`, `normal`, `numer`, `partfrac`, `polylib::decompose`, `polylib::divisors`, `polylib::primpart`, `polylib::sqrfree`, `rationalize`, `simplify`

Details:

- `factor` rewrites its argument as a product of as many terms as possible. In a certain sense, it is the complementary function of `expand`, which rewrites its argument as a sum of as many terms as possible.

- If `f` is a polynomial whose coefficient ring is not `Expr`, then `f` is factored over its coefficient ring. See example 8.

  If `f` is a polynomial with coefficient ring `Expr`, then `f` is factored over the smallest ring containing the coefficients. Mathematically, this *implied coefficient ring* always contains the ring $\mathbb{Z}$ of integers. See example 4.

  If the coefficient ring `R` of `f` is not `Expr`, then we say that the implied coefficient ring is `R`. Elements of the implied coefficient ring are considered to be constants and are not factored any further. In particular, the content u is an element of the implied coefficient ring.

- If `f` is an arithmetical expression that is not a number, it is considered as a rational expression. Non-rational subexpressions such as `sin(x)`, `exp(1)`, `x^(1/3)` etc., but not constant algebraic subexpressions such as `I` and `(sqrt(2)+1)^3`, are replaced by auxiliary variables before factoring. Algebraic dependencies of the subexpressions, such as the equation $\cos(x)^2 = 1 - \sin(x)^2$, are not necessarily taken into account. See example 6.

  The resulting expression is then written as a quotient of two polynomial expressions in the original and the auxiliary indeterminates. The numerator and the denominator are converted into polynomials with coefficient ring `Expr` via `poly`, and the implied coefficient ring is the smallest ring containing the coefficients of the numerator polynomial and the denominator polynomial. Usually, this is the ring of integers. Then both polynomials are factored over the implied coefficient ring, and the multiplicities e_i corresponding to factors of the denominator are negative integers; see example 3. After the factorization, the auxiliary variables are replaced by the original subexpressions. See example 5.

- If `f` is an integer, then it is decomposed into a product of primes, and the result is the same as for `ifactor`. If `f` is a rational number, then both the numerator and the denominator are decomposed into a product of primes. In this case, the multiplicities e_i corresponding to factors of the denominator are negative integers. See example 2.

- If `f` is a floating point number or a complex number, then `factor` returns a factorization with the single factor `f`.

- The result of `factor` is an object of the domain type `Factored`. Let `g:=factor(f)` be such an object.

  It is represented internally by the list `[u, f1, e1, ..., fr, er]` of odd length $2r + 1$. Here, `f1` through `fr` are of the same type as the input (either polynomials or expressions); `e1` through `er` are integers; and `u` is an arithmetical expression.

  One may extract the content u, the factors f_i, as well as the exponents e_i by the ordinary index operator `[ ]`, i.e., `g[1] = u, g[2] = f1, g[3] = e1, ...`.

  For example, to extract all irreducible factors of f, enter `g[2*i] $ i = 1..nops(g) div 2`. The same can be achieved with the call `Factored::factors(g)`, and the call `Factored::exponents(g)` returns a list of the exponents e_i for $1 \leq i \leq r$.

  The call `coerce(g,DOM_LIST)` returns the internal representation of a factored object, i.e., the list as described above.

  Note that the result of `factor` is printed as an expression, and it is implicitly converted into an expression whenever it is processed further by other MuPAD functions. As an example, the result of `q:=factor(x^2+2*x+1)` is printed as `(x+1)^2`, which is an expression of type `"_power"`.

See example 1 for illustrations, and the help page of Factored for details.

- If f is not a number, then each of the polynomials $p_1, ..., p_r$ is primitive, i.e., the greatest common divisor of its coefficients (see content and gcd) over the implied coefficient ring (see above for a definition) is one.
- Currently, factoring polynomials is possible over the following implied coefficient rings: integers and rational numbers, finite fields—represented by IntMod(n) or Dom::IntegerMod(n) for a prime number n, or by a Dom::GaloisField—, and rings obtained from these basic rings by taking polynomial rings (see Dom::DistributedPolynomial, Dom::MultivariatePolynomial, Dom::Polynomial, and Dom::UnivariatePolynomial), fields of fractions (see Dom::Fraction), and algebraic extensions (see Dom::AlgebraicExtension). In particular, factoring over the real and over complex numbers is *not* possible.
- If the input f is an arithmetical expression that is not a number, all occurring floating point numbers are replaced by continued fraction approximations. The result is sensitive to the environment variable DIGITS, see numeric::rationalize for details.

Example 1. To factor the polynomial $x^3 + x$, enter:

```
>> g := factor(x^3+x)

                                 2
                            x (x   + 1)
```

Usually, expressions are factored over the ring of integers, and factors with non-integral coefficients, such as x - I in the example above, are not considered.

One can access the internal representation of this factorization with the ordinary index operator:

```
>> g[1];                              // the content
   g[2*i]     $ i = 1..nops(g) div 2; // the factors
   g[2*i + 1] $ i = 1..nops(g) div 2; // the exponents

                                  1

                                  2
                             x, x   + 1

                                1, 1
```

The internal representation of g, as described above, is given by the following command:

```
>> coerce(g, DOM_LIST)
```

```
                    2
        [1, x, 1, x  + 1, 1]
```

The result of the factorization is an object of domain type `Factored`:

```
>> domtype(g)

                   Factored
```

Some of the functionality of this domain is described in what follows.

One may extract the factors and exponents of the factorization also in the following way:

```
>> Factored::factors(g), Factored::exponents(g)

                      2
                 [x, x  + 1], [1, 1]
```

One can ask for the type of factorization:

```
>> Factored::getType(g)

                 "irreducible"
```

This output means that all f_i are irreducible. Other possible types are `"squarefree"` (see `polylib::sqrfree`) or `"unknown"`.

One may multiply factored objects, which preserves the factored form:

```
>> g2 := factor(x^2 + 2*x + 1)

                           2
                   (x + 1)

>> g * g2

                     2              2
                x (x  + 1) (x + 1)
```

It is important to note that one can apply (almost) any function working with arithmetical expressions to an object of type `Factored`. However, the result is then usually not of domain type `Factored`:

```
>> expand(g);
   domtype(%)

                          3
                     x + x

                   DOM_EXPR
```

For a detailed description of these objects, please refer to the help page of the domain `Factored`.

Example 2. `factor` splits an integer into a product of prime factors:

```
>> factor(8)

                                  3
                                 2
```

For rational numbers, both the numerator and the denominator are factored:

```
>> factor(10/33)

                                 2 5
                                ----
                                3 11
```

Note that, in contrast, constant polynomials are *not* factored:

```
>> factor(poly(8, [x]))

                                  8
```

Example 3. Factors of the denominator are indicated by negative multiplicities:

```
>> factor((z^2 - 1)/z^2)

                           (z + 1) (z - 1)
                           ---------------
                                   2
                                  z

>> Factored::factors(%), Factored::exponents(%)

                      [z, z + 1, z - 1], [-2, 1, 1]
```

Example 4. If some coefficients are irrational but algebraic, the factorization takes place over the smallest field extension of the rationals that contains all of them. Hence, `x^2+1` is considered irreducible while its `I`-fold is considered reducible:

```
>> factor(x^2 + 1), factor(I*x^2 + I)

                        2
                       x  + 1, I (x - I) (x + I)
```

MuPAD cannot factor over the field of algebraic numbers; only the coefficients of the input are adjoined to the rationals:

```
>> factor(sqrt(2)*x^4 - sqrt(2)*x^2 - sqrt(2)*2)

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

>> factor(I*x^4 - I*x^2 - I*2)

                                         2
                    I (x - I) (x + I) (x   - 2)

>> factor(sqrt(2)*I*x^4 - sqrt(2)*I*x^2 - sqrt(2)*I*2)

              1/2                 1/2        1/2
          (I 2   ) (x + I) (x + 2   ) (x - 2   ) (x - I)
```

Example 5. Transcendental objects are treated as indeterminates:

```
>> delete x:
   factor(7*(exp(x)^2 - 1)*sin(1)^3)

                                                    3
                7 (exp(x) + 1) (exp(x) - 1) sin(1)

>> Factored::factors(%), Factored::exponents(%)

            [exp(x) + 1, exp(x) - 1, sin(1)], [1, 1, 3]
```

Example 6. `factor` regards transcendental subexpressions as algebraically independent of each other. Hence the binomial formula is not applied in the following example:

```
>> factor(x + 2*sqrt(x) + 1)

                             1/2
                       x + 2 x    + 1
```

Example 7. `factor` replaces floating point numbers by continued fraction approximations, factors the resulting polynomial, and finally applies `float` to the coefficients of the factors:

```
>> factor(x^2 + 2.0*x - 8.0)

                      (x + 4.0) (x - 2.0)
```

Example 8. Polynomials with a coefficient ring other than `Expr` are factored over their coefficient ring. We factor the following polynomial modulo 17:

```
>> R := Dom::IntegerMod(17): f:= poly(x^3 + x + 1, R):
   factor(f)

 poly(x + 6, [x], Dom::IntegerMod(17))

          2
    poly(x  + 11 x + 3, [x], Dom::IntegerMod(17))
```

For every `p`, the expression `IntMod(p)` may be used instead of `Dom::IntegerMod(p)`:

```
>> R := IntMod(17): f:= poly(x^3 + x + 1, R):
   factor(f)

                                    2
 poly(x + 6, [x], IntMod(17)) poly(x  - 6 x + 3, [x], IntMod(17)

    )
```

Example 9. More complex domains are allowed as coefficient rings, provided they can be obtained from the rational numbers or from a finite field by iterated construction of algebraic extensions, polynomial rings, and fields of fractions. In the following example, we factor the univariate polynomial $u^2 - x^3$ in u over the coefficient field $F = \mathbb{Q}(x, \sqrt{x})$:

```
>> Q := Dom::Rational:
   Qx := Dom::Fraction(Dom::DistributedPolynomial([x], Q)):
   F := Dom::AlgebraicExtension(Qx, poly(z^2 - x, [z])):
   f := poly(u^2 - x^3, [u], F)

       2    3
 poly(u  - x , [u], Dom::AlgebraicExtension(

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

                                           2
    Dom::Rational, LexOrder)), - x + z  = 0, z))

>> factor(f)

 poly(u - x z, [u], Dom::AlgebraicExtension(

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

                                           2
```

```
Dom::Rational, LexOrder)), - x + z  = 0, z)) poly(u + x z,

[u], Dom::AlgebraicExtension(Dom::Fraction(

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

       2
- x + z  = 0, z))
```

Background:

- The factoring algorithms are collected in a separate library domain `faclib`; it should not be necessary to call these routines directly.
- The implemented algorithms include Cantor-Zassenhaus (over finite fields) and Hensel lifting (over the rational numbers and in the multivariate case).

`fclose` – close a file

`fclose(n)` closes the file specified by the file descriptor `n`.

Call(s):

- `fclose(n)`

Parameters:

`n` — a file descriptor returned by `fopen`: a positive integer

Return Value: the void object of type `DOM_NULL`.

Related Functions: `fileIO`, `FILEPATH`, `finput`, `fname`, `fopen`, `fprint`, `fread`, `ftextinput`, `import::readdata`, `pathname`, `print`, `protocol`, `read`, `readbytes`, `READPATH`, `write`, `writebytes`, `WRITEPATH`

Details:

- The file must have been opened with `fopen`. The call to `fopen` yields the file descriptor `n` representing the file.
- Only a limited number of file descriptors is available. The user should use `fclose` to close a file which is no longer needed because this releases the file descriptor. The exact number of file descriptors available depends on the used operating system.

- For an overview of all file related MuPAD functions, also try `?fileIO`.
- `fclose` is a function of the system kernel.

Example 1. We open a file `test` for writing. This yields the file descriptor `n`:

```
>> n := fopen("test", Write)

                                   16
```

We close the file:

```
>> fclose(n): delete n:
```

`fileIO` – an overview of MuPAD's file I/O functions

There is a variety of functions in MuPAD to open, read, write, and close files. On this page, you get an overview of these functions.

Call(s):

- `fclose(..)`
- `finput(..)`
- `fname(..)`
- `fopen(..)`
- `fprint(..)`
- `fread(..)`
- `ftextinput(..)`
- `import::readdata(..)`
- `pathname(..)`
- `protocol(..)`
- `read(..)`
- `readbytes(..)`
- `write(..)`
- `writebytes(..)`
- `FILEPATH`
- `READPATH`
- `WRITEPATH`

Details:

- The following types of files can be used with MuPAD:

File type		MuPAD functions

	all file types	`fopen`
		`fclose`
		`fname`

ASCII	file with MuPAD commands	`finput`
		`fread`
		`read`
		`write`

	formatted text data	`fprint`
		`fread`
		`ftextinput`
		`import::readdata`
		`read`

	session information	`protocol`

Binary	MuPAD binary files	`finput`
		`fread`
		`read`
		`write`

	'raw' binary data	`readbytes` `writebytes`

MuPAD ASCII files (extension `.mu`) are ordinary text files containing MuPAD commands. For example, they are used to store all the shipped library code.

With the functions `fread` and `read`, the user can read in files containing MuPAD commands, i.e., programs written in the MuPAD programming language.

With 'formatted text data' we mean that the file contains, e.g., a sequence of numbers, strings or similar. Such files can be used to store intermediate results, or to create files that can be processed with an editor.

The `protocol` function enables the user to create a file that contains all inputs and outputs made during the session.

MuPAD binary files are mostly used for internal purposes. The user can use them to save and restore the state of some or all variables.

The functions `readbytes` and `writebytes` for dealing with 'raw' binary data enable you to read and write arbitrary files and interpret their contents as a sequence of numbers. For example, you can read image files, sound files etc. and use MuPAD's mathematical abilities to compress or encrypt the file.

- **Accessing files:** For most functions, there are two possibilities to access a file: either by directly specifying the filename as a string, or by opening the file via `fopen`, and passing the returned file descriptor.

  A file opened with `fopen` needs to be closed with `fclose`.

- **Controlling file location:** The functions `fopen`, `fprint`, `protocol`, `write`, and `writebytes` are sensitive to the environment variable `WRITEPATH`. If this variable has a value, the file is created in the corresponding directory. Otherwise, the file is created in the "working directory".

  The functions `finput`, `fopen`, `fread`, `read`, and `readbytes` are sensitive to the environment variable `READPATH`. First, the file is searched in the "working directory". If it can not be found there, all paths in `READPATH` are searched.

  Note that the meaning of "working directory" depends on the operating system. On Windows systems, the "working directory" is the folder where MuPAD is installed. On UNIX or Linux systems, it is the current working directory in which MuPAD was started.

- `fopen` opens a file for reading or writing. There are flags determining in which mode to open the file.

- `fclose` is used to close a file that was previously opened with `fopen`.

- `fname` returns the name of a file previously opened via `fopen`.

- `finput` reads MuPAD objects from a file and assigns them to identifiers.

- `fread` reads and executes a file containing MuPAD commands.

- `read` reads and executes a file containing MuPAD commands.

- `write` writes the values of some or all identifiers to a file.

- `fprint` writes ASCII data to a file.

- `ftextinput` reads an ASCII file, interpreting each line as one string.

- `import::readdata` is used to read formatted data from an ASCII file.

- `protocol` creates a protocol of your MuPAD session: All inputs and—optionally—outputs are stored in a file.

- `readbytes` reads any (binary) file interpreting the contents as numbers.

- `writebytes` writes a list of MuPAD numbers to a binary file.

- `pathname` is used to combine folder names to a platform dependent path name.

- `FILEPATH` is a variable containing the path to a file.

- `READPATH` determines where to search for files to be opened with `finput`, `fopen`, `fread`, `read`, and `readbytes`.
- `WRITEPATH` specifies where to open/create new files when calling `fopen`, `fprint`, `protocol`, `write` and `writebytes`.

`finput` – read MuPAD objects from a file

`finput(filename, x)` reads a MuPAD object from a file and assigns it to the identifier `x`.

`finput(n, x)` reads from the file associated with the file descriptor `n`.

Call(s):

- `finput(filename)`
- `finput(filename, x1, x2, ...)`
- `finput(n)`
- `finput(n, x1, x2, ...)`

Parameters:

`filename`	—	the name of a file: a character string
`n`	—	a file descriptor provided by `fopen`: a positive integer
`x1, x2, ...`	—	identifiers

Return Value: the last object that was read from the file.

Related Functions: `fclose`, `fileIO`, `fname`, `fopen`, `fprint`, `fread`, `ftextinput`, `input`, `loadproc`, `pathname`, `print`, `protocol`, `read`, `READPATH`, `textinput`, `write`, `WRITEPATH`

Details:

- `finput` can read MuPAD binary files as well as ASCII text files. `finput` recognizes the format of the file automatically.

  Binary files may be created via `fprint` or `write`. Text files can also be created in a MuPAD session via these functions (using the ***`Text`*** option; see the corresponding help pages for details). Alternatively, text files can be created and edited directly using your favourite text editor. The file must consist of syntactically correct MuPAD objects or statements, separated by semicolons or colons. An object may extend over more than one line.

- `finput(filename)` reads the first object in the file and returns it to the MuPAD session.

- `finput(filename, x1, x2, ...)` reads the contents of a file object by object. The i-th object is assigned to the identifier x_i. The identifiers are not evaluated while executing `finput`; previously assigned values are overwritten. The objects are not evaluated. Evaluation can be enforced with the function `eval`. Cf. example 2.
- Instead of a file name, also a file descriptor `n` of a file opened via `fopen` can be used. The functionality is as described above. However, there is one difference: With a file name, the file is closed automatically after the data were read. A subsequent call to `finput` starts at the beginning of the file. With a file descriptor, the file remains open (use `fclose` to close the file). The next time data are read from this file, the reading continues at the current position. Consequently, a file descriptor should be used if the individual objects in the file are to be read via several subsequent calls of `finput`. Cf. example 3.
- If the number of identifiers specified in the `finput` call is larger than the number of objects in the file, the exceeding identifiers are not assigned any values. In such a case, `finput` returns the void object of type `DOM_NULL`.
- `finput` interprets the file name as a pathname relative to the "working directory".

  Note that the meaning of "working directory" depends on the operating system. On Windows systems, the "working directory" is the folder where MuPAD is installed. On UNIX or Linux systems, it is the current working directory in which MuPAD was started.

  On the Macintosh, an empty file name may be given. In this case, a dialogue box is opened in which the user can choose a file.

  Also absolute path names are processed by `finput`.
- Expression sequences are not flattened by `finput` and cannot be used to pass several identifiers to `finput`. Cf. example 4.
- For an overview of all file related MuPAD functions, also try `?fileIO`.
- `finput` is a function of the system kernel.

Example 1. We write the numbers 11, 22, 33 and 44 into a file:

```
>> fprint("test", 11, 22, 33, 44):
```

We read this file with `finput`:

```
>> finput("test", x1, x2, x3, x4)
```

44

```
>> x1, x2, x3, x4
```

```
11, 22, 33, 44
```

If we try to read more objects than stored in the file, `finput` returns the void object of type `DOM_NULL`:

```
>> finput("test", x1, x2, x3, x4, x5); domtype(%)

                    DOM_NULL

>> x1, x2, x3, x4, x5

               11, 22, 33, 44, x5

>> delete x1, x2, x3, x4:
```

Example 2. Objects read from a file are not evaluated:

```
>> fprint("test", x1): x1 := 23: finput("test")

                       x1

>> eval(%)

                       23

>> delete x1:
```

Example 3. We read some data from a file using several calls of `finput`. We have to use a file descriptor for reading from the file. The file is opened for reading with `fopen`:

```
>> fprint("test", 11, 22, 33, 44): n := fopen("test"):
```

The file descriptor returned by `fopen` can be passed to `finput` for reading the data:

```
>> finput(n, x1, x2): x1, x2

                     11, 22

>> finput(n, x3, x4): x3, x4

                     33, 44
```

Finally, we close the file and delete the identifiers:

```
>> fclose(n): delete n, x1, x2, x3, x4:
```

Alternatively, the contents of a file can be read into a MuPAD session in the following way:

```
>> n := fopen("test"):
   for i from 1 to 4 do
      x.i := finput(n)
   end_for:
   x1, x2, x3, x4

                         11, 22, 33, 44

>> fclose(n): delete n, i, x1, x2, x3, x4:
```

Example 4. Expression sequences are not flattened by `finput` and cannot be used to pass identifiers to `finput`:

```
>> fprint("test", 11, 22, 33): finput("test", (x1, x2), x3)

 Error: Illegal argument [finput]
```

The following call does not lead to an error because the identifier `x12` is not evaluated. Consequently, only one object is read from the file and assigned to `x12`:

```
>> x12 := x1, x2: finput("test", x12): x1, x2, x12

                           x1, x2, 11

>> delete x12:
```

`float` – convert to a floating point number

`float(object)` converts the object or numerical subexpressions of the object to floating point numbers.

Call(s):

- `float(object)`

Parameters:

`object` — any MuPAD object

Return Value: a floating point number of type `DOM_FLOAT` or `DOM_COMPLEX`, or the input object with exact numbers replaced by floating point numbers.

Overloadable by: `object`

Side Effects: The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Related Functions: `DIGITS`, `Pref::floatFormat`, `Pref::trailingZeroes`

Details:

- `float` converts numbers and numerical expressions such as `sqrt(sin(2))` or `sqrt(3) + sin(PI/17)*I` to real or complex floating point numbers of type `DOM_FLOAT` or `DOM_COMPLEX`, respectively. If symbolic objects other than the special constants `CATALAN`, `E`, `EULER`, and `PI` are present, only *numerical* subexpressions are converted to floats. In particular, identifiers and indexed identifiers are returned unchanged by `float`. Cf. example 1.

- A `float` call is mapped recursively to the operands of an expression. When numbers (or constants such as `PI`) are found, they are converted to floating point approximations. The number of significant decimal digits is given by the environment variable `DIGITS`; the default value is 10. The converted operands are combined by arithmetical operations or function calls according to the structure of the expression. E.g., a call such as `float(PI - 314/100)` may be regarded as a sequence of numerical operations:

  ```
  t1 := float(PI); t2 := float(314/100); result := t1 - t2
  ```

  Consequently, float evaluation via `float` may be subject to error propagation. Cf. example 2.

- `float` is automatically mapped to the elements of sets and lists. However, it is not automatically mapped to the entries of arrays, tables, and operands of function calls. Use `map(object, float)` for a fast floating point conversion of all entries of an array or a table. Use `mapcoeffs(p, float)` to convert the coefficients of a polynomial `p` of type `DOM_POLY`. To control the behavior of `float` on a function call, use a function environment providing a `"float"` slot. Cf. examples 3 and 4.

- The preferences `Pref::floatFormat` and `Pref::trailingZeroes` can be used to modify the screen output of floating point numbers.

- Rational approximations of floating point numbers may be computed by the function `numeric::rationalize`.

- MuPAD's special functions such as `sin`, `exp`, `besselJ` etc. are implemented as function environments. Via overloading, the `"float"` attribute (slot) of a function environment `f`, say, is called for the float evaluation of symbolic calls `f(x1, x2, ...)` contained in an expression.

The user may extend the functionality of the system function `float` to his own functions. For this, the function `f` to be processed must be declared as a function environment via `funcenv`. A `"float"` attribute must be written, which is called by the system function `float` in the form `f::float(x1, x2, ...)` whenever a symbolic call `f(x1, x2, ...)` inside an expression is found. The arguments passed to `f::float` are not converted to floats, neither is the return value of the slot subject to any further float evaluation. Thus, the float conversion of symbolic functions calls of `f` is entirely determined by the slot routine. Cf. example 4.

- Also a domain `d`, say, written in the MuPAD language, can overload `float` to define the float evaluation of its elements. A slot `d::float` must be implemented. If an element `x`, say, of this domain is subject to a float evaluation, the slot is called in the form `d::float(x)`. As for function environments, neither `x` nor the return value of the slot are subject to any further float evaluation.

  If a domain does not have a `"float"` slot, the system function `float` returns its elements unchanged.

- Note that MuPAD's floating point numbers are restricted in size. On 32 bit architectures, an overflow/underflow occurs if numbers of absolute size larger/smaller than about $10.0^{\pm 2\,525\,222}$ are encountered. On 64 bit architectures, the limits are about $10.0^{\pm 42\,366\,205\,509\,363}$.

- See the documentation for `DIGITS` for further information.

- `float` is a function of the system kernel.

Example 1. We convert some numbers and numerical expressions to floats:

```
>> float(17), float(PI/7 + I/4), float(4^(1/3) + sin(7))

           17.0, 0.4487989505 + 0.25 I, 2.244387651
```

`float` is sensitive to `DIGITS`:

```
>> DIGITS := 20:
   float(17), float(PI/7 + I/4), float(4^(1/3) + sin(7))

  17.0, 0.44879895051282760549 + 0.25 I, 2.2443876506869885651
```

Symbolic objects such as identifiers are returned unchanged:

```
>> DIGITS := 10: float(2*x + sin(3))

                    2.0 x + 0.1411200081
```

Example 2. We illustrate error propagation in numerical computations. The following rational number approximates `exp(2)` to 17 decimal digits:

```
>> r := 738905609893065023/100000000000000000:
```

The following `float` call converts `exp(2)` and `r` to floating point approximations. The approximation errors propagate and are amplified in the following numerical expression:

```
>> DIGITS := 10: float(10^20*(r - exp(2)))

                              320.0
```

None of the digits in this result is correct! A better result is obtained by increasing `DIGITS`:

```
>> DIGITS := 20: float(10^20*(r - exp(2)))

                       276.95725394785404205

>> delete r, DIGITS:
```

Example 3. `float` is mapped to the elements of sets and lists:

```
>> float([PI, 1/7, [1/4, 2], {sin(1), 7/2}])

  [3.141592654, 0.1428571429, [0.25, 2.0], {0.8414709848, 3.5}]
```

For tables and arrays, the function `map` must be used to forward `float` to the entries:

```
>> T := table("a" = 4/3, 3 = PI): float(T), map(T, float)

                table(        table(
                  3 = PI,   ,   3 = 3.141592654,
                  "a" = 4/3     "a" = 1.333333333
                )             )

>> A := array(1..2, [1/7, PI]): float(A), map(A, float)

           +-       -+  +-                           -+
           | 1/7, PI |, | 0.1428571429, 3.141592654 |
           +-       -+  +-                           -+
```

Matrix domains overload the function `float`. In contrast to arrays, `float` works directly on a matrix:

```
>> float(matrix(A))
```

```
+-              -+
|  0.1428571429  |
|                |
|   3.141592654  |
+-              -+
```

Use `mapcoeffs` to apply `float` to the coefficients of a polynomial generated by `poly`:

```
>> p := poly(9/4*x^2 + PI, [x]): float(p), mapcoeffs(p, float)

             2                         2
    poly(9/4 x  + PI, [x]), poly(2.25 x  + 3.141592654, [x])

>> delete A, T, p:
```

Example 4. We demonstrate overloading of `float` by a function environment. The following function `Sin` is to represent the sine function. In contrast to MuPAD's `sin`, it measures its argument in degrees rather than in radians (i.e., `Sin(x)` = `sin(PI/180*x)`). The only functionality of `Sin` is to produce floating point values if the argument is a real float. For all other kinds of arguments, a symbolic function call is to be returned:

```
>> Sin := proc(x)
          begin
            if domtype(x) = DOM_FLOAT then
                 return(Sin::float(x));
            else return(procname(args()))
            end_if;
         end_proc:
```

The function is turned into a function environment via `funcenv`:

```
>> Sin := funcenv(Sin):
```

Finally, the `"float"` attribute is implemented. If the argument can be converted to a real floating point number, a floating point result is produced. In all other cases, a symbolic call of `Sin` is returned:

```
>> Sin::float := proc(x)
                 begin x := float(x):
                       if domtype(x) = DOM_FLOAT then
                            return(float(sin(PI/180*x)));
                       else return(Sin(x))
                       end_if;
                 end_proc:
```

Now, float evaluation of arbitrary expressions involving `Sin` is possible:

```
>> Sin(x), Sin(x + 0.3), Sin(120)

                Sin(x), Sin(x + 0.3), Sin(120)

>> Sin(120.0), float(Sin(120)), float(Sin(x + 120))

          0.8660254038, 0.8660254038, Sin(x + 120.0)

>> float(sqrt(2) + Sin(120 + sqrt(3)))

                         2.264730594

>> delete Sin:
```

---

fname – get a file name

`fname(n)` returns the name of the file specified by the file descriptor `n`.

Call(s):

- `fname(n)`

Parameters:

`n` — a file descriptor returned by `fopen`: a positive integer

Return Value: the name of the file: a character string of type `DOM_STRING`, or `NIL`.

Related Functions: `fclose`, `fileIO`, `finput`, `fopen`, `fprint`, `fread`, `ftextinput`

Details:

- The file must have been opened with `fopen`. The call to `fopen` yields the file descriptor `n` representing the file.
- The special file descriptor `0` represents no file but output to the user interface instead; `fname(0)` returns `NIL`.
- See `?fileIO` for a survey of all MuPAD functions for reading and writing files.
- `fname` is a function of the system kernel.

Example 1. We open a temporary file for writing. This yields the file descriptor `n`:

```
>> n := fopen(TempFile);
```

$$16$$

We get the file's name. Note that the name depends on the operating system:

```
>> fname(n);
```

"/tmp/mtxM9fPT"

Changes:

- `fname` is a new function.

`fopen` – open a file

`fopen(filename)` opens the file with the name `filename`.

Call(s):

- `fopen(filename <, mode> <, format>)`

Parameters:

`filename` — the name of a file: a character string or the flag *TempFile*

Options:

`mode` — either *Read*, *Write* or *Append*. With *Read*, the file is opened for reading; with *Write* or *Append*, it is opened for writing. If a file opened for writing does not yet exist, it is created. With *Write*, existing files are overwritten. With *Append*, new data may be appended to an existing file. Note that in the *Append* mode, the specified format must coincide with the format of the existing file; otherwise, the file cannot be opened and `fopen` returns `FAIL`. If the flag *TempFile* is given, the default mode is *Write*. Otherwise, the default mode is *Read*.

`format` — the write format: either *Bin*, *Text* or *Raw*. With *Bin*, the data are stored in MuPAD's internal binary format. With *Text*, the data may be strings or MuPAD objects stored as text. Newlines are handled according to the conventions of the operating system at hand. With *Raw*, the data must be binary maschine numbers. See the functions `readbytes` and `writebytes`.
If the mode is *Read* or *Append*, the default is the format of the data in the existing file. If the mode is *Write*, the default is *Bin*.

Return Value: a positive integer: the file descriptor. `FAIL` is returned if the file cannot be opened.

Side Effects: The function is sensitive to the environment variable `WRITEPATH` when creating files that are not temporary (temporary files are created via *TempFile*). If `WRITEPATH` has a value, in write mode (using the options *Write* or *Append*), the file is created in the corresponding directory. Otherwise, the file is created in the "working directory". A temporary file is created in a special directory.

Related Functions: `fclose`, `fileIO`, `FILEPATH`, `finput`, `fname`, `fprint`, `fread`, `ftextinput`, `pathname`, `print`, `protocol`, `read`, `readbytes`, `READPATH`, `write`, `writebytes`, `WRITEPATH`

Details:

- `fopen(filename <, Read>, format)` opens an existing file for reading in the specified format. An error is raised if no file with the specified name is found or the format of the file does not coincide with the specified `format`.

- `fopen(filename <, Read>)` opens an existing file for reading. The file must hold data in text or MuPAD's binary format, `fopen` automatically identifies the file format in this case. The file must no be used as raw file.

- `fopen(filename, mode, format)` opens the file for writing in the specified format if the mode is given as *Read* or *Append*. If no file with the specified name exists, a new file is created.

- `fopen(`*TempFile* <`,` *Write*> <`, format`>`)` creates and opens a temporary file for writing in the specified format. The option *Read* and *Append* are not allowed in this case. If no format is given, *Bin* is used. Use `fname` to query the actual name and location of the temporary file. Cf. example 3.

- In write mode (using one of the options *Write* or *Append*), the environment variable `WRITEPATH` is considered if no temporary file is created. If it has a value, a new file is created (or an existing file is searched for) in the corresponding directory. Otherwise, it is created/searched for in the "working directory".

  Note that the "working directory" depends on the operating system. On Windows systems, it is the folder, where MuPAD is installed. On UNIX or Linux systems, the "working directory" is the directory where MuPAD was started.

  In read mode, `fopen` does not search for files in the directories given by `READPATH` and `LIBPATH`. NOTE

  A temporary file is created in a special directory. This directory and the name of the file are system dependent.

  On the Macintosh, an empty file name may be given. In this case, a dialog box is opened in which the user can choose a file. Further, on the interactive level, MacMuPAD warns the user if an existing file is about to be overwritten.

  Also absolute path names are processed by `fopen`.

- The file descriptor returned by `fopen` can be used by various functions such as `fname`, `fclose`, `fread`, `fprint`, `read`, `write` etc.

- A file opened by `fopen` should be closed by `fclose` after use. This holds also for temporary files.

- `fopen` accepts its arguments in any order, not only in the order used above.

- For an overview of all file related MuPAD functions, also try `?fileIO`.

- `fopen` is a function of the system kernel.

Example 1. We open the file `test` for writing. With the option *Write*, it is not necessary that the file `test` exists. By default, the file is opened as a binary file:

```
>> n := fopen("test", Write)
```

16

We write a string to the file and close it:

```
>> fprint(n, "a string"): fclose(n):
```

We append another string to the file:

```
>> n := fopen("test", Append)
```

17

```
>> fprint(n, "another string"): fclose(n):
```

The binary file cannot be opened as a text file for appending data:

```
>> n := fopen("test", Append, Text)
```

FAIL

However, it may be opened as a text file with the option *`Write`*. The existing binary file is overwritten as a text file:

```
>> n := fopen("test", Write, Text)
```

18

```
>> fclose(n): delete n:
```

Example 2. `fopen` fails to open non-existing files for reading. Here we assume that the file `xyz` does not exist:

```
>> n := fopen("xyz")
```

FAIL

We assume that the file `test` created in example 1 exists. It can be opened for reading successfully:

```
>> n := fopen("test")
```

19

```
>> fclose(n): delete n:
```

Example 3. We open a temporary file, write 10 binary data bytes into it and close it. `fname` is used to query the name of the file:

```
>> fd := fopen(TempFile, Raw):
   writebytes(fd, [i $ i=1..10]):
   fn := fname(fd):
   fclose(fd):
   fn
```

```
                    "/tmp/mupad.7aYAp4"
```

Now, we re-open the file and read the data byte by byte:

```
>> fd := fopen(fn, Read, Raw):
   l := []:
   repeat
       r := readbytes(fd, 2);
       l := l.r;
   until r = [] end:
   l
```

```
                 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
>> fclose(fd): delete fd, fn, l:
```

Changes:

- New options *TempFile*, *Read* and *Raw* were introduced.
- The arguments can now be given in any order.

`for` – for loop

`for - end_for` is a repetition statement providing a loop for automatic iteration over a range of numbers or objects.

Call(s):

- ```
  for i from start to stop <step stepwidth> do
     body
  end_for
  ```
- `_for(i, start, stop, stepwidth, body)`

- ```
  for i from start downto stop <step stepwidth> do
    body
  end_for
  ```

- `_for_down(i, start, stop, stepwidth, body)`

- <u>`for`</u> `x` <u>`in`</u> `object` <u>`do`</u>
  `body`
  <u>`end_for`</u>
- `_for_in(x, object, body)`

Parameters:

`i, x`	—	the loop variable: an identifier or a local variable (`DOM_VAR`) of a procedure
`start`	—	the starting value for `i`: a real number. This may be an integer, a rational number, or a floating point number.
`stop`	—	the stopping value for `i`: a real number. This may be an integer, a rational number, or a floating point number.
`stepwidth`	—	the step width: a positive real number. This may be an integer, a rational number, or a floating point number. The default value is 1.
`object`	—	an arbitrary MuPAD object
`body`	—	the body of the loop: an arbitrary sequence of statements

Return Value: the value of the last command executed in the body of the loop. If no command was executed, the value `NIL` is returned. If the iteration range is empty, the void object of type `DOM_NULL` is returned.

Further Documentation: Chapter 16 of the MuPAD Tutorial.

Related Functions: `break`, `next`, `repeat`, `while`

Details:

- When entering an incrementing loop

  `for i from start to stop step stepwidth do body end_for`,

  the assignment `i := start` is made. The body is executed with this value of `i` (the body may reassign a new value to `i`). After all statements inside the body are executed, the loop returns to the beginning of the body, increments `i := i + stepwidth` and checks the stopping criterion `i > stop`. If `FALSE`, the body is executed again with the new value of `i`. If `TRUE`, the loop is terminated immediately without executing the body again.

- The decrementing loop

  `for i from start downto stop step stepwidth do body end_for` implements a corresponding behavior. The only difference is that upon return to the beginning of the body, the loop variable is decremented by `i := i - stepwidth` before the stopping criterion `i < stop` is checked.

- The loop `for x in object do body end_for` iterates `x` over all operands of the object. This loop is equivalent to

```
for i from 1 to nops(object) do
  x := op(object, i);
  body
end_for
```

  Typically, `object` may be a list, an expression sequence, or an array. Note that other container objects such as finite sets or tables do not have a natural internal ordering, i.e., care must be taken, if the loop expects a certain ordering of the iterative steps.

- The body of a loop may consist of any number of statements which must be separated either by a colon `:` or a semicolon `;`. The last evaluated result inside the body is printed on the screen as the return value of the loop. Use `print` inside the loop to see intermediate results.

- The loop variable `i`, respectively `x`, may have a value before the loop starts. After the loop is terminated, it has the value that was assigned in the last step of the loop. Typically, in an incrementing or decrementing loop with integer values of `start`, `stop`, and `stepwidth`, this is `i` = `stop` ± `stepwidth`.

- The arguments `start`, `stop`, `stepwidth`, and `object` are evaluated only once at the beginning of the loop and not after every iteration. E.g., if `object` is changed in a step of the loop, `x` still runs through all operands of the original object.

- Loops can be exited prematurely using the `break` statement. Steps of a loop can be skipped using the `next` statement. Cf. example 2.

- The keyword `end_for` may be replaced by the keyword `end`. Cf. example 3.

- Instead of the the imperative loop statements, the equivalent calls of the functions `_for`, `_for_down`, or `_for_in` may be used. Cf. example 4.

- `_for`, `_for_down` and `_for_in` are functions of the system kernel.

Example 1. The body of the following loop consists of several statements. The value of the loop variable `i` is overwritten when the loop is entered:

```
>> i := 20:
   for i from 1 to 3 do
     a := i;
     b := i^2;
     print(a, b)
   end_for:
```

```
                              1, 1

                              2, 4

                              3, 9
```

The loop variable now has the value that satisfied the stopping criterion `i > 3`:

```
>> i
```

```
                                4
```

The iteration range is not restricted to integers:

```
>> for i from 2.2 downto 1 step 0.5 do
     print(i)
   end_for:
```

```
                               2.2

                               1.7

                               1.2
```

The following loop sums up all elements in a list. The return value of the loop is the final sum. It can be assigned to a variable:

```
>> s := 0: S := for x in [c, 1, d, 2] do s := s + x end_for
```

```
                            c + d + 3
```

Note that for sets, the internal ordering is not necessarily the same as printed on the screen:

```
>> S := {c, d, 1}
```

```
                            {c, d, 1}
```

```
>> for x in S do print(x) end_for:
```

```
                                1

                                d

                                c
```

```
>> delete a, b, i, s, S, x:
```

Example 2. Loops can be exited prematurely using the `break` statement:

```
>> for i from 1 to 3 do
     print(i);
     if i = 2 then break end_if
   end_for:
```

```
                                1

                                2
```

With the `next` statement, the execution of commands in a step can be skipped. The evaluation continues at the beginning of the body with the incremented value of the loop variable:

```
>> a := 0:
   for i from 1 to 3 do
     a := a + 1;
     if i = 2 then next end_if;
     print(i, a)
   end_for:
```

```
                              1, 1

                              3, 3
```

```
>> delete i, a:
```

Example 3. Loops can be closed with the keyword `end` instead of `end_for`. The parser recognizes the scope of `end` statements automatically.

```
>> s:= 0:
   for i from 1 to 3 do
     for j from 1 to 3 do
       s := i + j;
       if i + j > 4 then
         break;
       end
     end
   end
```

```
                                5
```

```
>> delete s, i, j:
```

Example 4. This example demonstrates the correspondence between the functional and the imperative form of `for` loops:

```
>> hold(
     _for(i, start, stop, stepwidth, (statement1; statement2))
   )

           for i from start to stop step stepwidth do
             statement1;
             statement2
           end_for
```

The optional `step` clause is omitted by specifying the value `NIL` for the step width:

```
>> hold(
     _for_down(i, 10, 1, NIL, (x := i^2; x := x - 1))
   )

                    for i from 10 downto 1 do
                      x := i^2;
                      x := x - 1
                    end_for
```

```
>> hold(
     _for_in(x, object, body)
   )

                       for x in object do
                         body
                       end_for
```

`fprint` – write data to a file

`fprint(filename, objects)` writes MuPAD objects to the file `filename`.

`fprint(n, objects)` writes to the file associated with the file descriptor `n`.

Call(s):

- `fprint(<style,> <format,> filename <, object1, object2, ...>)`
- `fprint(<style,> n <, object1, object2, ...>)`

Parameters:

`filename`	—	the name of a file: a character string
`object1, object2, ...`	—	arbitrary MuPAD objects
`n`	—	a file descriptor provided by `fopen`: a nonnegative integer

Options:

`style`	—	either *Unquoted* or *NoNL*. These options are relevant for text files only. Both options make `fprint` store character strings without quotation marks. All objects are stored without separating colons in the text file. With *Unquoted*, a newline character is appended to the line generated by `fprint`. With *NoNL*, no newline character is appended to the line.
`format`	—	the write format: either *Bin* or *Text*. With *Bin*, the data are stored in MuPAD's binary format. With *Text*, standard ASCII format is used. The default is *Bin*.

Return Value: the void object of type `DOM_NULL`.

Side Effects: The function is sensitive to the environment variable `WRITEPATH`. If this variable has a value, the file is created in the corresponding directory. Otherwise, the file is created in the "working directory".

Related Functions: `expr2text`, `fclose`, `fileIO`, `finput`, `fname`, `fopen`, `fread`, `ftextinput`, `pathname`, `print`, `protocol`, `read`, `READPATH`, `write`, `WRITEPATH`

Details:

- `fprint` is used to write MuPAD objects to a file. The objects are evaluated, the results are stored in the file. These data can be read into another MuPAD session via the functions `finput` and `ftextinput`, respectively.

- The file may be specified directly by its name. In this case, `fprint` creates a new file or overwrites an existing file. `fprint` opens and closes the file automatically.

  If `WRITEPATH` does not have a value, `fprint` interprets the file name as a pathname relative to the "working directory".

  Note that the meaning of "working directory" depends on the operating system. On Windows systems, the "working directory" is the folder where MuPAD is installed. On UNIX or Linux systems, it is the current working directory in which MuPAD was started.

  On the Macintosh, an empty file name may be given. In this case, a dialogue box is opened in which the user can choose a file. Further, on the interactive level, MacMuPAD warns the user, if an existing file is about to be overwritten.

Also absolute path names are processed by `fprint`.

- Instead of a file name, also a file descriptor of a file opened via `fopen` can be used. Cf. example 2. In this case, the data written by `fprint` are appended to the corresponding file. The file is not closed automatically by `fprint` and must be closed by a subsequent call to `fclose`.

  Note that `fopen(filename)` opens the file in read-only mode. A subsequent `fprint` command to this file causes an error. Use the *Write* or *Append* option of `fopen` to open the file for writing.

  The file descriptor `0` represents the screen.

- Text output occurs without the Pretty-Printer. A call to `fprint` writes all specified objects into a single line of the text file. A newline character is appended to this line, unless the option *NoNL* is used. By default, the written objects are separated by colons without any further white space. The resulting text data consists of syntactically correct MuPAD code and can be read again using `finput`. With the options *Unquoted* and *NoNL*, neither white space no colons are inserted to separate the objects. The resulting text data cannot be read again using `finput`. Cf. example 3.

  NOTE

  Note that the text version of a MuPAD object does not necessarily reflect its data structure. A domain element stored in text mode may be read as an element of a different type by `finput`. Use the binary mode if stored data are to be read in their original form into another MuPAD session. Cf. example 4.

- MuPAD statements such as assignments etc. must be bracketed as in `fprint("test", (a := 2))`.

- For an overview of all file related MuPAD functions, also try `?fileIO`.

- `fprint` is a function of the system kernel.

Option <`Unquoted`>:

- This option is useful for writing user-formated text files. Data written with this option cannot be read again via `finput`.

- With this option, character strings are written without quotation marks. Additionally, the control characters `'\n'` and `'\t'` in strings are expanded. Furthermore, no colons are inserted between the objects. A newline character is appended to the line written by `fprint`.

Option <NoNL>:

- This option has the same functionality as *Unquoted*, with the only difference that no newline character is appended to the line written by `fprint`.

Example 1. We write some data to the file `test`. By default, this file is created as a binary file. For syntactical reasons, the assignment `d := 5` must be enclosed in additional brackets:

```
>> fprint("test", (d := 5), d*3):
```

The file is read into the MuPAD session. The assignment `d := 5` is executed, its return value is assigned to the identifier `e`. The value `d*3` is assigned to the identifier `f`:

```
>> finput("test", e, f): d, e, f;

                              5, 5, 15

>> delete d, e, f:
```

Example 2. We use a file descriptor to access the file `test`. Several calls to `fprint` append data to the file:

```
>> n := fopen("test", Write):
   fprint(n, (d := 5), d*3):
   fprint(n, "more data"):
```

Using a file descriptor, we have to call `fclose` to close the file:

```
>> fclose(n):
```

The file is read into the MuPAD session, assigning the stored values to the identifiers `e`, `f`, and `g`:

```
>> finput("test", e, f, g ): e, f, g;

                         5, 15, "more data"

>> delete n, d, e, f, g:
```

Example 3. With the option *Unquoted*, character strings are written without quotation marks:

```
>> fprint(Text, "test1", "Hello World!", MuPAD + 1):
   fprint(Unquoted, Text, "test2", "Hello World!", MuPAD + 1):
```

Now, the files `test1` and `test2` have the following content:

```
test1:
"Hello World!":MuPAD + 1:

test2:
Hello World!MuPAD + 1
```

We can use `finput` or `ftextinput` to read the data from the file:

```
>> finput("test1", a, b): a, b;

                    "Hello World!", MuPAD + 1

>> ftextinput("test2", c): c

                     "Hello World!MuPAD + 1"

>> delete a, b, c:
```

Example 4. The text version of a MuPAD object does not necessarily reflect its data structure. E.g., the function `matrix` creates matrices of domain type `Dom::Matrix()`. The text version, however, is an array:

```
>> fprint(Text, "test", matrix([1, 2])):
   finput("test")

            array(1..2, 1..1, (1, 1) = 1, (2, 1) = 2)
```

Use the binary mode to guarantee that stored objects can be read in their original form:

```
>> fprint("test", matrix([1, 2])):
   finput("test"); domtype(%)

                              +-   -+
                              |  1  |
                              |     |
                              |  2  |
                              +-   -+

                           Dom::Matrix()
```

Changes:

- `fprint` may now be called without objects to print.

`frac` – the fractional part of a number

`frac(x)` represents the "fractional part" `x-floor(x)` of the number `x`.

Call(s):

- `frac(x)`

Parameters:

`x` — an arithmetical expression

Return Value: an arithmetical expression.

Overloadable by: `x`

Side Effects: The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Related Functions: `floor`

Details:

- For complex arguments, `frac` is applied separately to the real and imaginary part.
- For real numbers, the value `x-floor(x)` represented by `frac(x)` is a number from the interval $[0, 1)$. For positive arguments, you may think of `frac` as truncating all digits before the decimal point.
- For integer arguments, `0` is returned. For rational arguments, a rational number is returned. For arguments that contain symbolic identifiers, symbolic function calls are returned. For floating point arguments or non-rational exact expressions, floating point values are returned.
- NOTE If the argument is a floating point number of absolute value larger than $10^{\texttt{DIGITS}}$, then the result is affected by internal non-significant digits! Cf. example 2.
- NOTE Exact numerical data that are neither integers nor rational numbers are approximated by floating point numbers. For such arguments, the result depends on the present value of `DIGITS`! Cf. example 3.

Example 1. We demonstrate the fractional part of real and complex numbers:

```
>> frac(1234), frac(123/4), frac(1.234)

                        0, 3/4, 0.234

>> frac(-1234), frac(-123/4), frac(-1.234)

                        0, 1/4, 0.766

>> frac(3/2 + 7/4*I), frac(4/3 + 1.234*I)

             1/2 + 3/4 I, 0.3333333333 + 0.234 I
```

The fractional part of a symbolic numerical expression is returned as a floating point value:

```
>> frac(exp(123)), frac(3/4*sin(1) + I*tan(3))

          0.7502040792, 0.6311032386 + 0.8574534569 I
```

Expressions with symbolic identifiers produce symbolic function calls:

```
>> frac(x), frac(sin(1) + x^2), frac(exp(-x))

                                  2
           frac(x), frac(sin(1) + x ), frac(exp(-x))
```

Example 2. Care should be taken when computing the fractional part of floating point numbers of large absolute value:

```
>> 10^13/3.0

                        3.333333333e12
```

Note that only the first 10 decimal digits are "significant". Further digits are subject to round-off effects caused by the internal binary representation. These "insignificant" digits can enter the fractional part:

```
>> frac(10^13/3.0)

                        0.3333332539
```

The mantissa of the next floating point number does not have enough digits to store "digits after the decimal point":

```
>> floor(10^25/9.0), ceil(10^25/9.0), frac(10^25/9.0)

   1111111111111111111081984, 1111111111111111111081984, 0.0
```

Example 3. Exact numerical expressions are converted to floating point numbers. Consequently, the present setting of `DIGITS` affects the result:

```
>> x := 10^30 - exp(30)^ln(10) + 1/3

                                                                ln(10)
        3000000000000000000000000000001/3 - exp(30)
```

Note that the exact value of this number is 1/3. Floating point evaluation can be subject to severe cancellation:

```
>> DIGITS := 20: frac(x)

                                    0.0
```

The floating point result is more accurate when a higher precision used:

```
>> DIGITS := 30: frac(x)

                  0.333334211260080337524414062 5

>> delete x, DIGITS:
```

`frame` – create a new frame, change to an existing frame

The statement `frame X` changes to frame `X`, creating it if necessary.

Call(s):

- `frame X`
- `_frame(X)`
- `frame ..`
- `_frame(..)`
- `frame A::B::C`
- `_frame(A::B::C)`
- `frame ..::A`
- `_frame(..::A)`

Parameters:

`X` — an identifier or a frame
`A, B, C` — frames

Return Value: a MuPAD object of type `DOM_FRAME`

Related Functions: `:=`, `_assign`, `assign`, `_delete`, `delete`, `slot`

Details:

- Frames are a very technical concept in MuPAD. Most users will have no need for using them.
- Frames are similar to name spaces in other programming languages such as `C++`.
- A frame is a *closed environment* inside a MuPAD session. *Closed* means:

  - Changing the value of an identifier inside a frame does not change the value of the identifier outside the environment. There are no effects on computations outside the frame environment.
  - Within a frame, the values of identifiers in other frames can be accessed without interfering with the local values in the current frame.

  Frames are useful as scratch pads for auxiliary calculations.
- From a practical point of view, frames inside a MuPAD session can be visualized as follows:

  - Frames form a rooted tree. One may think of a frame as a directory in a hierarchical file system.
  - One can change between frames as one can change between directories.
  - In a frame, identifiers may have a specific value valid only in this frame.
- An identifier can have different values in different frames. It can also inherit a value from another frame. Cf. example 8.
- Creating and changing between frames:

  - The command `frame A;` changes to the frame `A`. If this frame did not exist, it is created and assigned to the identifier `A` in the current frame. The return value of the command is the new frame. Cf. example 1.
  - `frame;` or `frame .;` do not change the frame. These commands just return the current frame. Cf. example 1.
  - `frame ..;` changes the current frame to the parent frame. The return value is the parent frame. Cf. example 1.

- `frame ::;` changes the current frame to the root frame (the top level frame in the frame hierarchy). The return value is the root frame. The parent frame of the root frame is again the root frame. Cf. example 1.
- It is also possible to change the current frame using a *frame path*. The frame path consists of the 'path delimiter' `::` and the 'path components'. The path components are existing frames or the special symbols `.` and `..`, where `.` represents the current frame and `..` represents the parent frame. The path delimiter `::` represents the root frame if a path begins with this symbol. The path component `.` is only allowed at the beginning of a frame path. Cf. example 2.
- Frame paths can only be used to change between existing frames. It is not possible to create a new frame using a frame path. Cf. example 2.

Deleting frames

- The command `delete A;` deletes the frame `A` and all of its subframes. Cf. example 6.
- Due to the fact that a frame is assigned to an identifier, it is possible to access an existing frame `A` through another identifier `B`, say. This way, one can still access the frame `A` even after deletion via `delete A`. However, deletion turns `A` into an *anonymous frame*. This means that no identifier has a specific value belonging to this frame. Cf. example 7.

Accessing variables

- With frames, there are two ways to access the value of an identifier: unqualified and qualified access.

  By *unqualified access* we mean the access to the value of an identifier by using its name or an 'unqualified frame path'. An 'unqualified frame path' is a frame path that does not start with `.`, `..` or `::`, respectively. A special search strategy is used to find the value. See below for further details.

  By *qualified access* we mean the access to the value of an identifier by specifying an *exact* frame path where to look for its value. A path is exact if and only if it starts with `.`, `..` or `::`, respectively.
- Unqualified read access:
  The search strategy for accessing the value of an identifier `x` is as follows: First, the value of `x` is searched in the current frame. If it has no value in the current frame, then its value is searched recursively in the parent frame until a value is found. Cf. examples 3 and

4. If `x` has no value in any of the searched frames, the expression specifying `x` is returned. Cf. examples 3 and 4.

- Unqualified write access:
  If a value is assigned to an identifier `x`, the value of `x` is set in the current frame or in the frame specified by the frame path. Cf. example 5.

- Qualified read access:
  Qualified access means that the frame in which the value of an identifier `x` is searched is exactly specified (the frame path starts with `.`, `..` or `::`, respectively). If `x` has a value in this frame, the value is returned. Otherwise, the expression specifying `x` is returned symbolically. Cf. example 8.

- Qualified write access:
  Qualified access means that the frame is exactly specified in which a new value is to be assigned to an identifier (the frame path starts with `.`, `..` or `::`, respectively). Cf. example 9.

Inside a frame, the command `delete x;` deletes the value of the identifier `x` *in this frame.*

NOTE

This means that, in this frame, the value of `x` is the identifier itself.

It is possible to use a frame path while specifing the identifier to be deleted. Cf. example 11.

From a technical point of view, using a frame path is a slot access by the function `slot`. At the beginning of the frame path, `.` is parsed as `_frame()`, `..` as `_parentFrame()` and `::` as `_rootFrame()`. Within the frame path, `..` is parsed as the slot named `".."`. Cf. example 13. The `slot` function is then responsible for finding the value for an identifier specified by the frame path as described above.

An identifier can have different values in different frames. It can also have different properties in different frames.

NOTE

It is not possible to use a frame path to specify an identifier for setting properties via `assume` or reading properties via `getprop`.

Cf. example 10.

NOTE

Domains have a unique data representation in the MuPAD system. Changing a method of a domain will change this method for all instances of the domain in the MuPAD system. It is not possible to do local changes to a domain that are valid for one frame only. Cf. example 12.

Example 1. In the following call, a new frame is created because the identifier `A` has no value:

```
>> delete A: frame A
```

```
frame ::A
```

The current frame is now the just created frame `A`:

```
>> frame
```

```
frame ::A
```

We change back to the root frame. The output `frame ::` denotes the root frame:

```
>> frame ..
```

```
frame ::
```

The identifier `A` contains the created frame. It is possible to change back to this frame:

```
>> A; frame A:
```

```
frame ::A
```

Instead of using `frame ..`, there exists another way to return to the root frame and there exist also an alternative to see the current frame:

```
>> frame ::; frame .
```

```
frame ::

frame ::
```

If an identifier already has a value which is no frame, it is also possible to use this identifier for creating a new frame, overwriting the old value. We set `B` to `42`:

```
>> B := 42
```

```
42
```

Now, the frame `B` is created which overrides the old value of `B`:

```
>> frame B: B
```

```
frame ::B
```

We change to the root frame and delete the created frames:

```
>> frame :: : delete A, B:
```

Example 2. A hierarchy of frames is created. First, frame `A` inside the root frame is created:

```
>> frame A
```

```
frame ::A
```

Frame `B` inside the frame `A` is created:

```
>> frame B
```

```
frame ::A::B
```

It is not possible to create a frame using a frame path:

```
>> frame ..::C
```

```
Error: Illegal argument [frame]
```

We create 2 frames `C`: one inside frame `A`, and another one inside the root frame. Before creating the frames, we have to move to the corresponding places in the frame hierarchy:

```
>> frame ..; frame C
```

```
frame ::A
```

```
frame ::A::C
```

```
>> frame ::; frame C
```

```
frame ::
```

```
frame ::C
```

We change to the frame `C` inside frame `A`:

```
>> frame ::A::C
```

```
frame ::A::C
```

We return to the root frame and delete all created frames:

```
>> frame :: : delete A, C:
```

Example 3. Frames are useful for having a so-called "closed environment": Changing the value of an identifier `x` in a frame will not influence the value of `x` in other frames. We now take a closer look at this behavior. Inside the root frame, the value of `x` is set to `42`:

```
>> x := 42:
```

Inside frame `A`, `x` is set to the string "value inside frame A". If `x` is accessed by its name, we get the value belonging to the current frame:

```
>> frame A: x := "value inside frame A":
   x
```

```
"value inside frame A"
```

In the root frame, `x` is not changed:

```
>> frame ..: x
```

```
42
```

The new frame `B` is created and no specific value is assigned to `x` inside this frame. While accessing `x`, the search strategy for finding its value is used. The value is found in the parent frame (which is the root frame):

```
>> frame B: x
```

```
42
```

The same holds if another subframe `C` is created. Again `x` has no value in this subframe and the value is found in the first parent frame where `x` has a value:

```
>> frame C: frame; x
```

```
frame ::B::C
```

```
42
```

If an identifier has no value in any frame the value is searched for, the identifier itself is returned:

```
>> y
```

```
y
```

We return to the root frame and delete the created frames:

```
>> frame :: : delete A, B:
```

Example 4. A similiar situation as in example 3 is created. But here frame `B` and `C` are subframes of `A`:

```
>> x := 42: frame A:
   x := "value inside frame A":
   frame B: frame C
```

```
frame ::A::B::C
```

The current frame is changed to the root frame and the value of `x` is searched in different frames. This is explained now in more detail. First, the value of `x` is searched which is found in the root frame:

```
>> frame ::; x
```

```
frame ::

42
```

Then, the value of `x` is searched in the frame `A`. To be more precise, the value of `A` is searched and found to be `frame ::A`. Then, the value of the slot named `x` is looked up in this frame. This is the value of `x` in the frame `A`, which exists:

```
>> A::x
```

```
"value inside frame A"
```

This also explains the next result. First, the value of `A` is searched. The result is the frame `::A` . Then, the value of the slot `B` is accessed which is the frame `::A::B`. Finally, the value of the `slot x` inside the frame `::A::B` is searched which is the value of the identifier `x` in the frame `::A::B` which does not exist. The search strategy for finding a value of an identifier is only used for the first identifier in a frame path. After that, the usual mechanism for a `slot` access is used.

The search strategy is *n*ot used recursively here:

```
>> A::B::x
```

```
A::B::x
```

The current frame is changed to the frame `A::B::C`. If `A::x` is accessed, the value of the identifier `A` is searched. The value of `A` is the frame `::A` found in the root frame. After `A` is found, the slot `x` is accessed which is the value of `x` inside the frame `::A`:

```
>> frame A::B::C: A::x
```

```
"value inside frame A"
```

We return to the root frame and delete the created frames:

```
>> frame :: : delete A:
```

Example 5. Now, a frame path is used to write the value of an identifier. First the value of `x` in the root frame is set:

```
>> x := 42:
```

The current frame is set to the root frame and the value of `x` belonging to the frame `A` is set to the string "value inside frame A". The technical explanation is similar to the read access using a frame path:

```
>> frame A: frame :: :
   A::x := "value inside frame A":
   x, A::x
```

```
                  42, "value inside frame A"
```

Even if we change the current frame to frame `A` and assign a value to `A::y`, this sets the value of `y` belonging to frame `A`. The reason is again the search strategy. First the value of `A` is searched. The result is the frame `A` belonging to the identifier `A` in the root frame. Then, the value of `y` inside this frame is set:

```
>> frame A: A::y := 12: y
```

```
                              12
```

We return to the root frame and delete all created frames:

```
>> frame :: : delete A:
```

Example 6. First, a frame hierarchy is created and some identifiers are assigned values valid only in particular frames:

```
>> x := 42 : frame A: y := 21: frame B: z := 1:
```

The identifier `A` is deleted. The frame `A` and all of its subframes are no longer available:

```
>> frame :: : delete A:
   A::y
```

```
 Error: Unknown slot "A::y" [slot]
```

If the current frame is changed to frame `A`, `A` is newly created. No identifier has a value in this frame:

```
>> frame A: y
```

```
                               y
```

Since the subframe `B` was deleted implicitly when deleting `A`, the next command creates a new instance of frame `B`. No identifier has a value in this frame:

```
>> frame B: z
```

```
                              z
```

We return to the root frame and delete all created frames:

```
>> frame :: : delete A:
```

Example 7. As in the last example, a frame hierarchy is created and some identifiers are assigned values valid only in special frames:

```
>> x := 42 : frame A: y := 21: frame B: z := 1:
```

Before the frame `A` is deleted, we assign its value to the identifier `C` in the root frame:

```
>> frame :: : C := A:
```

Now, the identifier `A` is deleted: the frame A and all of its subframes are no longer available. Due to the fact that `C` still contains the old frame `A`, we can still change to the old frame `A` by using `C`. Since `A` was deleted, it is now an "anonymous" frame:

```
>> delete A: frame C
```

```
                    frame ::AnonymousFrame
```

While it was still possible to change to the now anonymous frame `A`, all the identifiers which had a value in the old frame `A` lost their values:

```
>> B, y
```

```
                             B, y
```

We return to the root frame and delete all created frames:

```
>> frame :: : delete A:
```

Example 8. One may read the value of an identifier using an exact frame path. First, a frame hierarchy is created:

```
>> x := 42: frame A: x := 21: y := 13: frame B: x := 1:
```

Accessing the identifier `x` from the current frame `A::B` gives the result `1` because `x` is defined in the current frame. Getting the value of `x` from other exactly specified frames needs a qualified read access to these values:

```
>> x, ..::x, ::x
```

```
1, 21, 42
```

We present some other possibilities for accessing the value of `x` from frame `A` or the root frame. In the first case, the value of `x` is searched in the frame `A` which should be a subframe of the root frame. In the second case, it is searched in the parent frame of the parent frame:

```
>> ::A::x, ..::..::x
```

```
21, 42
```

If the value of `y` in the current frame should be accessed, it is not enough to use `y` because this would find the value of `y` in the parent frame `A`. To avoid this, `.` can be used which represents the current frame. The output of `.::y` is the unevaluated expression `(_frame())::y` because `y` has no value in the current frame and `.` is the operator notation for `_frame()`:

```
>> y, .::y
```

```
13, (_frame())::y
```

We return to the root frame and delete all created frames:

```
>> frame :: : delete A:
```

Example 9. In the last example, read access for the value of an identifier in an exactly specified frame was shown. Now the same is done for write access. First, a frame hierarchy is created:

```
>> frame A: frame B:
```

Now the same situation as in the last example should be created without leaving the current frame `B`. This means that the value of `x` in the root frame should be `42`, in frame `A` it should be `21` and in frame `B` it should be `1`. Also the value of `y` in frame `A` should be `13`:

```
>> x := 1: ..::x := 21: ::x := 42: ..::y := 13:
   x, ..::x, ::x, ..::y
```

```
1, 21, 42, 13
```

The following commands provide an alternative to assign these values:

```
>> .::x := 1: ::A::x := 21: ..::..::x := 42: ::A::y := 13:
   x, ..::x, ::x, ..::y
```

1, 21, 42, 13

We return to the root frame and delete all created frames:

```
>> frame :: : delete A:
```

Example 10. An identifier can have different properties in different frames:

```
>> delete x: assume(x > 0): getprop(x)
```

> 0

```
>> frame A: assume(x < 0): getprop(x)
```

< 0

In the root frame, `x` still has the property `> 0`:

```
>> frame .. : getprop(x)
```

> 0

It is not possible to use a frame path to specify an identifier for reading or writing its properties:

```
>> getprop(.::x), getprop(A::x)
```

(_frame())::x, A::x

```
>> assume(.::x = 0 )
```

```
Error: at least one side must be an (indexed) identifier [assu\
me]
```

We return to the root frame and delete all created frames:

```
>> frame :: : delete A:
```

Example 11. Deleting an identifier means deleting its value. After deletion, the identifier exists as a *symbol*. In the following, the identifier `x` has no value in the frame `A`; the value in the root frame is found:

```
>> x := 42: frame A : x

                              42
```

After `x` is deleted in the current frame, it has no value but exists as a symbol. Accessing the value of `x` returns the identifier itself:

```
>> delete x: x

                              x
```

Now, the value `11` is assigned to `y` in the root frame. This value is found in the current frame `A`:

```
>> ::y := 11: y

                              11
```

`y` is deleted in the current frame. Now, a frame path is used. As before, `y` exists as a symbol in the current frame `A`. Accessing the value of `y` returns the identifier itself:

```
>> delete ::A::y: y

                              y
```

We return to the root frame and delete all created frames:

```
>> frame :: : delete A:
```

Example 12. Domains have a unique data representation in the MuPAD system. Frames do not allow to change methods of a domain only in special frames. The new domain `"Test"` is created in the root frame and in the frame `A`. However, both domains have the same key `"Test"`. Hence, they are the same although they were created in different frames:

```
>> T := newDomain("Test"): frame A: TT := newDomain("Test"):
```

The print method of the domain `"Test"` in the frame `A` is defined: it returns the string "Operands" together with the operands of an element of the domain:

```
>> TT::print := elem -> return("Operands", op(elem))

              elem -> return("Operands", op(elem))
```

```
>> new(TT, 3)
```

"Operands", 3

Setting the print method in the frame `A` also set the print method of the domain inside the root frame, since both domains are the same:

```
>> frame :: : expose(T::print)
```

elem -> return("Operands", op(elem))

```
>> new(T, 7)
```

"Operands", 7

We return to the root frame and delete all created frames:

```
>> frame :: : delete A:
```

Example 13. The following example shows how the frame path is parsed when accessing the value of an identifier. First, a frame hierarchy is created:

```
>> delete x: frame A: frame B
```

frame ::A::B

Now, the value of `x` in the root frame is accessed. Since `x` has no value in the root frame, symbolic `slot` calls are returned:

```
>> ..::..::x; .::..::..::x; ::x
```

(slot(_parentFrame(), "..")):: x

(slot(slot(_frame(), ".."), "..")):: x

(_rootFrame()):: x

We return to the root frame and delete the created frames:

```
>> frame :: : delete A:
```

Changes:

- `frame` is a new keyword.

`frandom` – generate random floating point numbers

`frandom()` returns a pseudo-random floating point number from the interval $[0.0, 1.0)$.

`frandom(seed)` returns a generator of pseudo-random floating point numbers from the interval $[0.0, 1.0)$.

Call(s):

- `frandom()`
- `frandom(seed)`

Parameters:

`seed` — an initialization value for the generator: an integer

Return Value: `frandom()` returns a floating point number; `frandom(seed)` returns a procedure (a pseudo-random number generator).

Side Effects: `frandom` is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Related Functions: `random`, `stats::uniformRandom`

Details:

- The calls `frandom()` produce uniformly distributed floating point numbers from the interval $[0.0, 1.0)$.
- `r := frandom(seed)` produces a random number generator `r`. Subsequent calls `r()` return uniformly distributed floating point numbers from the interval $[0.0, 1.0)$. Different generators created with the same seed generate the same sequences of numbers. Cf. Example 3. Generators created in separate calls to `frandom` do not influence one another.
- As for all functions returning floating point numbers, `frandom` reacts to `DIGITS` and returns numbers with the precision set by this variable.
- Each time MuPAD is started or re-initialized with the `reset` function, random generators produce the same sequence of numbers.
- `frandom` is the recommended function for generating uniform random floating point numbers. It is much faster than the function `random` which produces uniform integer numbers.

In contrast to `random`, `frandom` does not react to the environment variable `SEED`.

- The function `stats::uniformRandom` allows to produce uniformly distributed floating point numbers on arbitrary finite intervals. The `stats` library also provides random generators with various other distributions.
- `frandom` is a function of the system kernel.

Example 1. The following call produces a sequence of pseudo-random numbers. Note that an index variable `i` must be used in the construction of the sequence. A call such as `frandom() $ 8` would produce 8 copies of the same random value:

```
>> frandom() $ i = 1..8

 0.2703567032, 0.8142678572, 0.1145977439, 0.247668289,

    0.436855213, 0.7507294917, 0.5143284818, 0.47002619
```

Example 2. `frandom` reacts to `DIGITS`:

```
>> DIGITS := 200: frandom(), frandom()

 0.069563338270290396129397095438205056004224903765787803686798\
 408761527212789416741722732042747764978052445739789905227204520\
 616931337224550761067946684014828343727527240325990111226020040\
 98029917237691462

    ,

 0.214621811832755486840726972406260492132624504839255676795843\
 025918749776185141686249484543634824878907064804516823735810680\
 412061656102457687820249754580614991121448596075708731127058980\
 6804671578401241

>> delete DIGITS:
```

Example 3. `frandom(seed)`, for some integer value of `seed`, returns a generator of floating point numbers. For different generators created with the same seed, the sequences of numbers will be identical (provided the value of `DIGITS` does not change):

```
>> r1 := frandom(42):
   r2 := frandom(42):
   r1() $ i=1..4;
   r2() $ i=1..4

     0.1105445771, 0.8801762635, 0.8463755466, 0.4128521752

     0.1105445771, 0.8801762635, 0.8463755466, 0.4128521752
```

Note that the sequences produced by these generators may differ when they are called with significantly different values of `DIGITS`:

```
>> r1 := frandom(42): r2 := frandom(42): r1() $ i=1..4

     0.1105445771, 0.8801762635, 0.8463755466, 0.4128521752

>> DIGITS := 20: r2() $ i=1..4

 0.11054457694991559l3, 0.26116556986774964568,

    0.92427601333877321223, 0.81737003913431684486

>> delete r1, r2, DIGITS:
```

Background:

- `frandom` uses a linear congruence generator to directly manipulate the internal representation of a `DOM_FLOAT`.

Changes:

- `frandom` is a new function.

`fread` – read and execute a file

`fread(filename)` reads and executes the MuPAD file `filename`.

`fread(n)` reads and executes the file associated with the file descriptor `n`.

Call(s):

- `fread(filename` <, *Quiet*> <, *Plain*>`)`
- `fread(n` <, *Quiet*> <, *Plain*>`)`

Parameters:

`filename`	—	the name of a file: a character string
`n`	—	a file descriptor provided by `fopen`: a positive integer

Options:

Plain — makes `fread` use its own parser context
Quiet — suppresses output during execution of `fread`

Return Value: the return value of the last statement of the file.

Related Functions: `fclose`, `fileIO`, `FILEPATH`, `finput`, `fname`, `fopen`, `fprint`, `ftextinput`, `input`, `loadproc`, `pathname`, `print`, `protocol`, `read`, `READPATH`, `textinput`, `write`, `WRITEPATH`

Details:

- `fread(filename)` reads the file and evaluates each MuPAD statement in the file.
- `fread` is similar to `read`. The only difference is that `fread` does not search for files in the directories given by `READPATH` and `LIBPATH`; `fread` only searches for the file relative to the "working directory".

  Note that the meaning of "working directory" depends on the operating system. On Windows systems, the "working directory" is the folder where MuPAD is installed. On UNIX or Linux systems, it is the current working directory in which MuPAD was started.

  On the Macintosh, an empty file name may be given. In this case, a dialogue box is opened in which the user can choose a file. Further, on the interactive level, MacMuPAD warns the user, if an existing file is about to be overwritten.

  Also absolute path names are processed by `fread`.
- `fread` can read MuPAD binary files (created via `fprint` or `write`) as well as ASCII text files. `fread` recognizes the format of the file automatically.
- Instead of a file name, also a file descriptor of a file opened via `fopen` can be used. Cf. example 3.
- Similar to procedures, the identifier `LEVEL` has the value 1 while the file's content is evaluated by `fread`.
- When a file is read with `fread` the variable `FILEPATH` contains the path of the file.
- `fread` is a function of the system kernel.

Option <`Quiet`>:

- With this option, output is suppressed while reading and executing the file. However, warnings, error messages as well as the output of `print` commands are still visible.

Option <Plain>:

- With this option, the file is read in a new parser context. This means that the `history` is not modified by the statements in the file. Further, abbreviations set outside the file via `alias` or user defined operators are ignored during the execution of the file. This option is useful for reading initialization files in a clean environment.

Example 1. The following example is only functional under UNIX and Linux; on other operating systems one must change the path names accordingly. First, we use `fprint` to create a file containing three MuPAD statements:

```
>> fprint(Unquoted, Text, "/tmp/test", "a := 3; b := 5; a + b;"):
```

When reading the file, the statements are executed. Each produces a print output. The second `8` below is the return value of `fread`:

```
>> delete a, b: fread("/tmp/test")

                              3

                              5

                              8

                              8
```

Now, the variables `a` and `b` have the values assigned inside the file :

```
>> a, b

                             3, 5
```

With the option *Quiet*, only the return value of `fread` is printed:

```
>> delete a, b: fread("/tmp/test", Quiet)

                              8

>> delete a, b:
```

Example 2. The next example demonstrates the option *Plain*. First, an appropriate input file is created:

```
>> fprint(Unquoted, Text, "/tmp/test",
          "f := proc(x) begin x^2 end_proc:",
          "a := f(3): b := f(4):"):
```

We define an `alias` for `f`:

```
>> alias(f = "some text"):
```

An error occurs if we try to read the file without the option *Plain*. In the parser context of the MuPAD session, the alias replaces `f` by the corresponding string in the assignment `f := ...`. However, strings cannot be assigned a value:

```
>> fread("/tmp/test"):

 Error: Invalid left-hand side [_assign];
 while reading file '/tmp/test'
```

With the option *Plain*, no such error arises: the alias for `f` is ignored by `fread`:

```
>> fread("/tmp/test", Plain): a, b

                              9, 16

>> unalias(f): delete f, a, b:
```

Example 3. We use `write` to save the value of the identifier `a` in the file "`/tmp/test`":

```
>> a := PI + 1: write("/tmp/test", a): delete a:
```

This file is opened for reading with `fopen`:

```
>> n := fopen("/tmp/test")

                               16
```

The file descriptor returned by `fopen` can be passed to `fread`. Reading the file restores the value of `a`:

```
>> fread(n): a

                             PI + 1

>> fclose(n): delete a:
```

`freeze`, `unfreeze` – create an inactive or active copy of a function

`freeze(f)` creates an inactive copy of the function `f`.

`unfreeze(object)` reactivates all inactive functions occurring in `object` and evaluates the result.

Call(s):

- `freeze(f)`
- `unfreeze(object)`

Parameters:

`f`	—	a procedure or a function environment
`object`	—	any MuPAD object

Return Value: `freeze` returns an object of the same type as `f`. `unfreeze` returns the evaluation of `object` after reactivating all inactive functions in it.

Related Functions: `eval`, `hold`, `MAXDEPTH`

Details:

- `ff := freeze(f)` returns a function that is an "inactive" copy of the argument `f`. This means:

  1. `ff` only evaluates its arguments, but does not compute anything else,
  2. `ff` is printed in the same way as `f`,
  3. symbolic `ff` calls have the same type as symbolic `f` calls,
  4. if `f` is a function environment, then `ff` has all the slots of `f`.

  Note that `ff` evaluates its incoming parameters even if the function `f` has the procedure option `hold`.

- `freeze` can be used when many operations with `f` are to be performed that require `f` only in its symbolic form, but `f` need not be executed. See example 2.
- Neither `eval` nor `level` can enforce the evaluation of an inactive function; see example 2.

- `unfreeze(object)` reactivates all inactive functions occurring in `object`, proceeding recursively along the structure of `object`, and then evaluates the result.
- `unfreeze` uses `misc::maprec` to proceed recursively along the structure of `object`. This means that for `basic domains` such as arrays, tables, lists, or polynomials, the function `unfreeze` is applied to each operand of `object`.

  Note that if `object` is an element of a `library domain`, then the behavior of `unfreeze` is specified by the method `"maprec"` which overloads the function `misc::maprec`. If this method does not exist, then `unfreeze` has no effect on `object`. See example 5.
- `unfreeze` does not operate on the body of procedures, therefore it is recommended not to embed inactive functions inside procedures.

Example 1. We create an inactive form of the function environment `int`:

```
>> _int := freeze(int): F := _int(x*exp(x^2), x = 0..1)

                                  2
                      int(x exp(x ), x = 0..1)
```

The inactive form of `int` keeps every information that is known about the function `int`, e.g., the output, the type, and the `"float"` slot for floating-point evaluation:

```
>> F, type(F), float(F)

                       2
          int(x exp(x ), x = 0..1), "int", 0.8591409142
```

The original function environment `int` is not modified by `freeze`:

```
>> int(x*exp(x^2), x = 0..1)

                             exp(1)
                             ------ - 1/2
                               2
```

Use `unfreeze` to reactivate the inactive function `_int` and evaluate the result:

```
>> unfreeze(F), unfreeze(F + 1/2)

                         exp(1)         exp(1)
                         ------ - 1/2, ------
                           2              2
```

Example 2. The function `student::riemann` makes use of `freeze` in order to return a result where the function `sum` is preserved in its symbolic form:

```
>> a:= student::riemann(sin(x), x = 0..PI)

          /    / PI (i3 + 1/2) \             \
   PI sum| sin| ------------- |, i3 = 0..3 |
          \    \       4       /             /
   -----------------------------------------
                       4
```

Only when applying `unfreeze` the sum is computed:

```
>> unfreeze(a)

            1/2      1/2            1/2 1/2
      PI ((2    + 2)    + (2 - 2   )   )
      ----------------------------------
                       4
```

```
>> float(%)

                    2.052344306
```

Example 3. We demonstrate the difference between `hold` and `freeze`. The result of the command `S := hold(sum)(...)` does not contain an inactive version of `sum`, but the unevaluated identifier `sum`:

```
>> S := hold(sum)(1/n^2, n = 1..infinity)

             /  1                   \
          sum| --, n = 1..infinity |
             |  2                   |
             \ n                    /
```

The next time `S` is evaluated, the identifier `sum` is replaced by its value, the function environment `sum`, and the procedure computing the value of the infinite sum is invoked:

```
>> S

                      2
                    PI
                    ---
                     6
```

In contrast, evaluation of the result of `freeze` does not lead to an evaluation of the inactive function:

```
>> S := freeze(sum)(1/n^2, n = 1..infinity)

                       /  1                  \
                    sum| --, n = 1..infinity |
                       |  2                  |
                       \ n                   /
```

```
>> S

                       /  1                  \
                    sum| --, n = 1..infinity |
                       |  2                  |
                       \ n                   /
```

An inactive function does not even react to `eval`:

```
>> eval(S)

                       /  1                  \
                    sum| --, n = 1..infinity |
                       |  2                  |
                       \ n                   /
```

The only way to undo a `freeze` is to use `unfreeze`, which reactivates the inactive function in `S` and then evaluates the result:

```
>> unfreeze(S)

                                  2
                                PI
                                ---
                                 6
```

Example 4. Note that `freeze(f)` does not change the object `f` but returns a copy of `f` in an inactive form. This means that computations with the inactive version of `f` may contain the original function `f`.

For example, if we create an inactive version of the sine function:

```
>> Sin := freeze(sin):
```

and expand the term `Sin(x+y)`, then the result is expressed in terms of the (original) sine function `sin`:

```
>> expand(Sin(x + y))

                    cos(x) sin(y) + cos(y) sin(x)
```

Example 5. The function unfreeze uses misc::maprec to operate recursively along the structure of object. For example, if object is an array containing inactive functions, such as:

```
>> a := array(1..2,
     [freeze(int)(sin(x), x = 0..2*PI), freeze(sum)(k^2, k = 1..n)]
   )

          +-                                2           -+
          | int(sin(x), x = 0..2 PI), sum(k , k = 1..n) |
          +-                                            -+
```

then unfreeze(a) operates on the operands of a:

```
>> unfreeze(a)

                         +-                  -+
                         |          2    3   |
                         |      n   n    n   |
                         |  0, - + -- + --   |
                         |      6   2    3   |
                         +-                  -+
```

This means that for library domains, the effect of unfreeze is specified by the method "maprec". If the domain does not implement this method, then unfreeze does not operate on the objects of this domain. For example, we create a dummy domain and an object containing an inactive function as its operand:

```
>> dummy := newDomain("dummy"):
   o := new(dummy, freeze(int)(sin(x), x = 0..2*PI))

                 new(dummy, int(sin(x), x = 0..2 PI))
```

The function unfreeze applied to the object o has no effect:

```
>> unfreeze(o)

                 new(dummy, int(sin(x), x = 0..2 PI))
```

If we overload the function misc::maprec in order to operate on the first operand of objects of the domain dummy, then unfreeze operates on o as desired:

```
>> dummy::maprec :=
     x -> extsubsop(x,
       1 = misc::maprec(extop(x,1), args(2..args(0)))
     ):
   unfreeze(o)

                         new(dummy, 0)
```

`ftextinput` – read a text file

`ftextinput(filename, x)` reads a line from a text file, interprets the line as a string and assigns this string to the identifier `x`.

`ftextinput(n, x)` reads from the file associated with the file descriptor `n`.

Call(s):

- `ftextinput(filename)`
- `ftextinput(filename, x1, x2, ...)`
- `ftextinput(n)`
- `ftextinput(n, x1, x2, ...)`

Parameters:

`filename`	—	the name of a file: a character string
`n`	—	a file descriptor provided by `fopen`: a positive integer
`x1, x2, ...`	—	identifiers

Return Value: the last line that was read from the file: a character string.

Related Functions: `fclose`, `fileIO`, `finput`, `fname`, `fopen`, `fprint`, `fread`, `input`, `pathname`, `print`, `protocol`, `read`, `READPATH`, `textinput`, `write`, `WRITEPATH`

Details:

- `ftextinput(filename)` reads the first line of the text file and returns it as a string to the MuPAD session.
- `ftextinput(filename, x1, x2, ...)` reads the file line by line. The i-th line is converted to a character string and assigned to the identifier x_i. The identifiers are not evaluated while executing `ftextinput`; previously assigned values are overwritten.
- Instead of a file name, also a file descriptor `n` of a file opened via `fopen` can be used. The functionality is as described above. However, there is one difference: With a file name, the file is closed automatically after the data were read. A subsequent call to `ftextinput` starts at the beginning of the file. With a file descriptor, the file remains open (use `fclose` to close the file). The next time data are read from this file, the reading continues at the current position. Consequently, a file descriptor should be used, if the individual lines in the file are to be read via several subsequent calls of `ftextinput`. Cf. example 2.

- If the number of identifiers specified in the `ftextinput` call is larger than the number of lines in the file, the exceeding identifiers are not assigned any values. In such a case, `ftextinput` returns the void object of type `DOM_NULL`.
- `ftextinput` interprets the file name as a pathname relative to the "working directory".

  Note that the meaning of "working directory" depends on the operating system. On Windows systems, the "working directory" is the folder where MuPAD is installed. On UNIX or Linux systems, it is the current working directory in which MuPAD was started.

  On the Macintosh, an empty file name may be given. In this case, a dialogue box is opened in which the user can choose a file.

  Also absolute path names are processed by `ftextinput`.
- Expression sequences are not flattened by `ftextinput` and cannot be used to pass several identifiers to `ftextinput`. Cf. example 3.
- For an overview of all file related MuPAD functions, also try `?fileIO`.
- `ftextinput` is a function of the system kernel.

Example 1. First, we use `fprint` to create a text file with three lines:

```
>> fprint(Unquoted, Text, "test", "x + 1\n2nd line\n3rd line"):
```

We read the first two lines of the file and assign the corresponding strings to the identifiers `x1` and `x2`:

```
>> ftextinput("test", x1, x2): x1, x2

                      "x + 1", "2nd line"
```

If we try to read beyond the last line of the file, `ftextinput` returns the void object of type `DOM_NULL`:

```
>> ftextinput("test", x1, x2, x3, x4); domtype(%)


                           DOM_NULL

>> x1, x2, x3, x4

              "x + 1", "2nd line", "3rd line", x4

>> delete x1, x2, x3:
```

Example 2. We read some lines from a file using several calls of `ftextinput`. We have to use a file descriptor for reading from the file. The file is opened for reading with `fopen`:

```
>> fprint(Unquoted, Text, "test",
          "x + 1\nx + 2\n3rd line\n4th line"):
```

```
>> n := fopen("test"):
```

The file descriptor returned by `fopen` can be passed to `ftextinput` for reading the data:

```
>> ftextinput(n, x1, x2): x1, x2

                        "x + 1", "x + 2"
```

```
>> ftextinput(n, x3, x4): x3, x4

                     "3rd line", "4th line"
```

Finally, we close the file and delete the identifiers:

```
>> fclose(n): delete n, x1, x2, x3, x4:
```

Alternatively, the contents of a file can be read into a MuPAD session in the following way:

```
>> n := fopen("test"):
   for i from 1 to 4 do
      x.i := ftextinput(n)
   end_for:
   x1, x2, x3, x4

            "x + 1", "x + 2", "3rd line", "4th line"
```

```
>> fclose(n): delete n, i, x1, x2, x3, x4:
```

Example 3. Expression sequences are not flattened by `ftextinput` and cannot be used to pass identifiers to `ftextinput`:

```
>> fprint(Unquoted, Text, "test", "1st line\n2nd line\n3rd line"):
   ftextinput("test", (x1, x2), x3)

 Error: Illegal argument [ftextinput]
```

The following call does not lead to an error because the identifier `x12` is not evaluated. Consequently, only one line is read from the file and assigned to `x12`:

```
>> x12 := x1, x2: ftextinput("test", x12): x1, x2, x12

                        x1, x2, "1st line"

>> delete x12:
```

---

funcenv – create a function environment

`funcenv` creates a function environment. A function environment behaves like an ordinary function with the additional possibility to define function attributes. These are used to overload standard system functions such as **`diff`**, **`float`** etc.

Call(s):

- `funcenv(f1 <, f2> <, slotTable>)`

Parameters:

`f1`	—	an arbitrary MuPAD object. Typically, a procedure. It handles the evaluation of a function call to the function environment.
`f2`	—	a procedure handling the screen output of symbolic function calls
`slotTable`	—	a table of function attributes (slots)

Return Value: a function environment of type `DOM_FUNC_ENV`.

Further Documentation: Chapter "Function Environments" of the Tutorial.

Related Functions: `slot`

Details:

- **`funcenv`** serves for generating a function environment of domain type `DOM_FUNC_ENV`.

  From a user's point of view, function environments are similar to procedures and can be called like any MuPAD function.

  However, in contrast to simple procedures, a function environment allows a tight integration into the MuPAD system. In particular, standard system functions such as **`diff`**, **`expand`**, **`float`** etc. can be told how to act on symbolic function calls to a function environment.

  For this, a function environment stores special function attributes (slots) in an internal table. Whenever an overloadable system function such as **`diff`**, **`expand`**, **`float`** encounters an object of type `DOM_FUNC_ENV`, its searches the function environment for a corresponding slot. If found, it calls the corresponding slot and returns the value produced by the slot.

Slots can be incorporated into the function environment by creating a table `slotTable` and passing this to `funcenv`, when the function environment is created. Alternatively, the function `slot` can be used to add further slots to an existing function environment.

See example 1 below for further information.

- The first argument `f1` of `funcenv` determines the evaluation of function calls. With `f:= funcenv(f1)`, the call `f(x)` returns the result `f1(x)`. Note that calls of the form `f:= funcenv(f)` are possible (and, in fact, typical). This call embeds the procedure `f` into a function environment of the same name. The original procedure `f` is stored internally in the function environment `f`. After this call, further function attributes can be attached to `f` via the `slot` function.

- The second argument `f2` of `funcenv` determines the screen output of symbolic function calls. Consider `f:= funcenv(f1, f2)`. If the call `f(x)` returns a symbolic function call `f(x)` with 0-th operand `f`, then `f2` is called: the return value of `f2(f(x))` is used as the screen output of `f(x)`.

  Beware: `f2(f(x))` should not produce a result containing a further symbolic call of `f`, because this will lead to an infinite recursion, causing an error message.

- The third argument `slotTable` of `funcenv` is a table containing function attributes (slots). The table has to use strings as indices to address system functions. E.g.,

  ```
  slotTable := table("diff" = mydiff, "float" = myfloat):
  f := funcenv(f1, f2, slotTable):
  ```

  attaches the slot functions `mydiff` and `myfloat` to `f`. They are called by the system functions `diff` and `float`, respectively, whenever they encounter a symbolic expression `f(x)` with 0-th operand `f`. The internal slot table can be changed or filled with additional function attributes via the function `slot`.

- The documentation of `float`, `print`, and `slot` provides further examples involving function environments.

- `funcenv` is a function of the system kernel.

Example 1. We want to introduce a function `f` that represents a solution of the differential equation $f'(x) = x + \sin(x)\, f(x)$. First, we define a function `f`, which returns any call `f(x)` symbolically:

```
>> f := proc(x) begin procname(args()) end_proc: f(x), f(3 + y)
```

```
f(x), f(y + 3)
```

Because of the differential equation $f'(x) = x + \sin(x)\, f(x)$, derivatives of `f` can be rewritten in terms of `f`. How can we tell the MuPAD system to differentiate symbolic functions calls such as `f(x)` accordingly? For this, we first have to embed the procedure `f` into a function environment:

```
>> f := funcenv(f):
```

The function environment behaves like the original procedure:

```
>> f(x), f(3 + y)

                    f(x), f(y + 3)
```

System functions such as `diff` still treat symbolic calls of `f` as calls to unknown functions:

```
>> diff(f(x + 3), x)

                      D(f)(x + 3)
```

However, as a function environment, `f` can receive attributes that overload the system functions. The following `slot` call attaches a dummy `"diff"` attribute to `f`:

```
>> f := slot(f, "diff", mydiff): diff(2*f(x^2) + x, x)

                              2
                 2 mydiff(f(x ), x) + 1
```

We attach a more meaningful `"diff"` attribute to `f` that is based on $f'(x) = x + \sin(x)\, f(x)$. Note, that arbitrary calls `diff(f(y), x1, x2, ..)` have to be handled by this slot:

```
>> fdiff := proc(fcall) local y; begin
       y:= op(fcall, 1);
       (y + sin(y)*f(y))*diff(y, args(2..args(0)))
   end_proc:
   f := slot(f, "diff", fdiff):
```

Now, as far as differentiation is concerned, the function `f` is fully integrated into MuPAD:

```
>> diff(f(x), x), diff(f(x), x, x)

   x + f(x) sin(x), f(x) cos(x) + sin(x) (x + f(x) sin(x)) + 1

>> diff(sin(x)*f(x^2), x)

                   2                  2      2       2
         cos(x) f(x ) + 2 x sin(x) (x  + f(x ) sin(x ))
```

Since Taylor expansion around finite points only needs to evaluate derivatives, also Taylor expansions of f can be computed:

```
>> taylor(f(x^2), x = 0, 9)

             4 / f(0)         \     8 / f(0)         \       9
     f(0) + x  | ---- + 1/2 | + x  | ---- + 1/8 | + O(x )
               \   2          /      \  12          /

>> delete f, fdiff:
```

Example 2. Suppose that you have defined a function f that may return itself symbolically, and you want such symbolic expressions of the form f(x,...) to be printed in a special way. To this end, embed your procedure f in a function environment and supply an output procedure as second argument to the corresponding funcenv call. Whenever an expression of the form f(x,...) is to be printed, the output procedure will be called with the arguments x,... of the expression:

```
>> f := funcenv(f,
            proc(x) begin
               if nops(x) = 2 then
                 "f does strange things with its arguments ".
                 expr2text(op(x, 1))." and ".expr2text(op(x,2))
               else
                 FAIL
               end
            end):

>> delete a, b:
   f(a, b)/2;
   f(a, b, c)/2

        f does strange things with its arguments a and b
        ------------------------------------------------
                                2

                            f(a, b, c)
                            ----------
                                2

>> delete f:
```

Example 3. For all prefedined `function environments`, the second operand is a built-in output function, of type `DOM_EXEC`. In particular, this is the case for operators such as `+`, `*`, `^` etc. In the following example, we change the output symbol for the power operator `^`, which is stored in the third operand of the built-in output function of the function environment `_power`, to a double asterisk:

```
>> unprotect(_power):
   _power := subsop(_power, [2, 3] = "**"):
   a^b/2;
   print(a^b/2):
   _power := subsop(_power, [2, 3] = "^"):
   protect(_power):
```

```
                     a**b
                     ----
                       2

                     a**b
                     ----
                       2
```

Background:

- Mathematical functions such as `exp`, `ln` etc. or `abs`, `Re`, `Im` etc. are implemented as function environments.

`gamma` – the gamma function

`gamma(x)` represents the gamma function $\Gamma(x) = \int_0^\infty e^{-t}\, t^{x-1}\, \mathrm{d}t$.

Call(s):

- `gamma(x)`

Parameters:

`x` — an arithmetical expression or a floating point interval

Return Value: an arithmetical expression or a floating point interval.

Overloadable by: `x`

Side Effects: When called with a floating point argument, the function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Related Functions: `beta`, `binomial`, `fact`, `igamma`, `psi`

Details:

- The gamma function is defined for all complex arguments apart from the singular points $0, -1, -2, \ldots$.
- It is related to the factorial function: `gamma(x)=fact(x-1)=(x-1)!` for all positive integers x.
- If `x` is a floating point value, then a floating point value is returned. If `x` is a floating point interval, a floating point interval is returned. If `x` is a positive integer smaller than 1000, then an integer is returned. If `x` is a rational number of domain type `DOM_RAT` satisfying `1<x<500`, then the functional relation $\Gamma(x+1) = x\,\Gamma(x)$ is applied to "normalize" the result. The functional relation

  $$\Gamma(x)\Gamma(1-x) = \frac{\pi}{\sin(\pi x)}$$

  is applied if `x<1/2` is a rational number of domain type `DOM_RAT` that is an integer multiple of 1/4 or 1/6. The call `gamma(1/2)` yields `sqrt(PI)`; `gamma(infinity)` yields `infinity`.

  For all other arguments, a symbolic function call is returned.
- The float attribute of `gamma` is a kernel function, i.e., floating point evaluation is fast.
- The logarithmic derivative of `gamma` is implemented by the digamma function `psi`.
- The `expand` attribute uses the functional equation $\Gamma(x+1) = x\,\Gamma(x)$, the reflection formula

  $$\Gamma(-x) = \frac{-\pi}{x\sin(\pi x)\Gamma(x)},$$

  and the Gauß multiplication formula for $\Gamma(kx)$ when k is a positive integer, to rewrite `gamma(x)`. Cf. example 3. For numerical `x`, the functional equation is used to shift the argument to the range `0 < x < 1`.

Example 1. We demonstrate some calls with exact and symbolic input data:

```
>> gamma(15), gamma(23/2), gamma(sqrt(2)), gamma(x + 1)

                                  1/2
                   13749310575 PI              1/2
    87178291200, -----------------, gamma(2   ), gamma(x + 1)
                        2048
```

Floating point values are computed for floating point arguments:

```
>> gamma(11.5), gamma(2.0 + 10.0*I)

      11899423.08, - 0.00001089258677 + 0.00000504737724 I
```

Example 2. `gamma` is singular for nonpositive integers:

```
>> gamma(-2)

 Error: singularity [gamma]
```

Example 3. The functions `diff`, `expand`, `float`, `limit`, and `series` handle expressions involving `gamma`:

```
>> diff(gamma(x^2 + 1), x), float(ln(3 + gamma(sqrt(PI))))

                  2              2
           2 x psi(x  + 1) gamma(x  + 1), 1.367203476

>> expand(gamma(3*x - 4))

                  1/2  3 x
        gamma(x) 3    3    gamma(x + 1/3) gamma(x + 2/3)
        -----------------------------------------------
          6 PI (3 x - 1) (3 x - 2) (3 x - 3) (3 x - 4)

>> limit(1/gamma(x), x = infinity),
   limit(gamma(x - 4)/gamma(x - 10), x = 0)

                           0, 151200

>> series(gamma(x), x = 0, 3)

                         /         2        2 \
                         |       PI    EULER  |
                         |     - --- + ------ |
        1                | PI      12      2   |      2
        - - EULER + x PI | -- + -------------- | + O(x )
        x                \  6          PI      /
```

The Stirling formula is obtained as an asymptotic series:

```
>> series(gamma(x), x = infinity, 4)
```

```
  1/2  x  1/2      1/2  x  1/2      1/2  x  1/2
PI    x  2       PI    x  2       PI    x  2
------------ + ------------- + -------------- +
  1/2               3/2               5/2
 x    exp(x)    12 x    exp(x)   288 x    exp(x)

     /       x     \
     |      x      |
    O| ----------- |
     |  7/2        |
     \ x    exp(x) /
```

Changes:

- Floating point intervals are now handled.
- The system function `expand` now implements the reflection formula and the Gauß multiplication formula.

`gcd` – the greatest common divisor of polynomials

`gcd(p, q, ...)` returns the greatest common divisor of the polynomials $p, q, \ldots$

Call(s):

- `gcd(p, q, ...)`
- `gcd(f, g, ...)`

Parameters:

`p, q, ...` — polynomials of type `DOM_POLY`
`f, g, ...` — polynomial expressions

Return Value: a polynomial, a polynomial expression, or the value `FAIL`.

Overloadable by: `p, q, f, g`

Related Functions: `content`, `div`, `divide`, `factor`, `gcdex`, `icontent`, `ifactor`, `igcd`, `igcdex`, `ilcm`, `lcm`, `mod`, `poly`

Details:

- `gcd(p, q, ...)` calculates the greatest common divisor of any number of polynomials. The coefficient ring of the polynomials may either be the integers or the rational numbers, *Expr*, a residue class ring *IntMod*(n) with a prime number n, or a domain.

  All polynomials must have the same indeterminates and the same coefficient ring.

- Polynomial expressions are converted to polynomials. See `poly` for details. `FAIL` is returned if an argument cannot be converted to a polynomial.

- The return value is of the same type as the input polynomials, i.e., either a polynomial of type `DOM_POLY` or a polynomial expression.

- `gcd` returns 0 if all arguments are 0, or if no argument is given. If at least one of the arguments is -1 or 1, then `gcd` returns 1.

- Use `igcd` if all arguments are known to be integers, since it is much faster than `gcd`.

Example 1. The greatest common divisor of two polynomial expressions can be computed as follows:

```
>> gcd(6*x^3 + 9*x^2*y^2, 2*x + 2*x*y + 3*y^2 + 3*y^3)

                                       2
                             2 x + 3 y

>> f := (x - sqrt(2))*(x^2 + sqrt(3)*x-1):
   g := (x - sqrt(2))*(x - sqrt(3)):
   gcd(f, g)

                                    1/2
                              x - 2
```

One may also choose polynomials as arguments:

```
>> p := poly(2*x^2 - 4*x*y - 2*x + 4*y, [x, y], IntMod(17)):
   q := poly(x^2*y - 2*x*y^2, [x, y], IntMod(17)):
   gcd(p, q)

                 poly(x - 2 y, [x, y], IntMod(17))

>> delete f, g, p, q:
```

Background:

- If the arguments are polynomials with coefficients from a domain, then the domain must have the methods `"gcd"` and `"_divide"`. The method `"gcd"` must return the greatest common divisor of any number of domain elements. The method `"_divide"` must divide two domain elements. If domain elements cannot be divided, this method must return `FAIL`.

`gcdex` – the extended Euclidean algorithm for polynomials

`gcdex(p, q, x)` regards `p` and `q` as univariate polynomials in `x` and returns their greatest common divisor as a linear combination of `p` and `q`.

Call(s):

- `gcdex(p, q <, x>)`
- `gcdex(f, g, x)`

Parameters:

`p, q`	—	polynomials of type `DOM_POLY`
`f, g`	—	polynomial expressions
`x`	—	an indeterminate: an identifier or an indexed identifier

Return Value: a sequence of three polynomials, or a sequence of three polynomial expressions, or `FAIL`.

Overloadable by: `p, q`

Related Functions: `factor`, `div`, `divide`, `gcd`, `ifactor`, `igcd`, `igcdex`, `ilcm`, `lcm`, `mod`, `poly`

Details:

- `gcdex(p, q, x)` returns a sequence g, s, t with three elements, where the polynomial g is the greatest common divisor of p and q. The polynomials s and t satisfy $g = s\,p + t\,q$ and $\deg(s) < \deg(q)$, $\deg(t) < \deg(p)$. These data are computed by the extended Euclidean algorithm.
- `gcdex` only processes univariate polynomials:
    - If the indeterminate `x` is specified, the input polynomials are regarded as univariate polynomials in `x`.
    - If no indeterminate is specified, the indeterminate of the polynomials is searched for internally. An error occurs if more than one indeterminate is found.

Note that x must be specified if polynomial expressions are used on input.

- Polynomial expressions are converted to polynomials. See `poly` for details. `FAIL` is returned if an argument cannot be converted to a polynomial.
- The returned polynomials are polynomial expressions if the input consists of polynomial expressions. Otherwise, polynomials of type `DOM_POLY` are returned.
- The coefficient ring of the polynomials must provide the method `"_divide"`. This method must return `FAIL` if domain elements cannot be divided.
- If the coefficient domain of the polynomial is not a field, then it may not be possible to represent a greatest common divisor as a linear combination of the input polynomials. In such a case, an error is raised.

Example 1. The greatest common divisor of two univariate polynomials in extended form can be computed as follows:

```
>> gcdex(poly(x^3 + 1), poly(x^2 + 2*x + 1))

   poly(x + 1, [x]), poly(1/3, [x]), poly(- 1/3 x + 2/3, [x])
```

For multivariate polynomials, an indeterminate must be specified:

```
>> gcdex(poly(x^2*y), poly(x + y), x)

                    /  1      \      / /    1 \     1      \
   poly(1, [x]), poly| --, [x] |, poly| | - -- | x + -, [x] |
                    |  3      |      | |    2 |     y      |
                    \ y       /      \ \   y  /            /

>> gcdex(poly(x^2*y), poly(x + y), y)

                          /    1      \      / 1      \
         poly(1, [y]), poly| - --, [y] |, poly| -, [y] |
                          |    3      |      \ x      /
                          \   x       /

>> gcdex(x^3 + a, x^2 + 1, x)

                                   2
                    a + x  1 - x  - a x
                 1, ------, ------------
                     2          2
                    a  + 1     a  + 1
```

`genident` – create an unused identifier

`genident()` creates an identifier not used before in the current session.

Call(s):

- `genident()`
- `genident(S)`

Parameters:

`S` — a character string

Return Value: an identifier.

Related Functions: `delete`, `hold`

Details:

- `genident()` creates an identifier with a name of the form `Xi`, where `i` is a positive integer. It is guaranteed that the returned identifier has not been used before in the current MuPAD session.
- If a string `S` is given as argument, then `genident` returns an identifier with a name of the form `Si`, where `i` is a positive integer.
- The returned identifier does not have a value.
- `genident` is a function of the system kernel.

Example 1. We create three new identifiers. The second identifier has a different prefix:

```
>> genident(), genident("Y"), genident()

                    X1, Y1, X2
```

In the next example, we assign a value to the identifier `X4`. Then the next two calls to `genident` skip the name `X4`:

```
>> X4 := 5:
   genident(), genident()

                    X3, X5
```

`genpoly` – create a polynomial using the "b"-adic expansion

`genpoly(n, b, x)` creates a polynomial p in the indeterminate x such that $p(b) = n$.

Call(s):

- `genpoly(n, b, x)`

Parameters:

- `n` — an integer, a polynomial of type `DOM_POLY`, or a polynomial expression
- `b` — an integer greater than 1
- `x` — the indeterminate: an identifier

Return Value: a polynomial if the first argument is a polynomial or an integer. Otherwise, a polynomial expression.

Related Functions: `genident`, `indets`, `int2text`, `interpolate`, `mods`, `numlib::g_adic`, `poly`, `text2int`

Details:

- `genpoly(n, b, x)` creates a polynomial `p` in the variable `x` from the `b`-adic expansion of `n`, such that $p(b) = n$. The integer coefficients of the resulting polynomial are greater than $-b/2$ and less than or equal to $b/2$.
- The b-adic expansion of an integer n is defined by $n = \sum_{i=0}^{m} c_i b^i$, such that the c_i are symmetric remainders modulo b, i.e., $-b/2 < c_i \le b/2$ for all i (see `mods`). From this expansion the polynomial $p = \sum_{i=0}^{m} c_i x^i$ is created. The polynomial is defined over the coefficient ring `Expr`.
- If the first argument of `genpoly` is a (multivariate) polynomial, then it must be defined over the coefficient ring `Expr` and must have only integer coefficients. The third argument `x` must not be a variable of the polynomial. In this case each integer coefficient is converted into a polynomial in `x` as described above. The result is a polynomial in the variable `x`, followed by the variables of the given polynomial. (`x` is the main variable of the returned polynomial.)
- The first argument `n` may also be a polynomial expression. In this case, it is converted into a polynomial using `poly`, then `genpoly` is applied as described above, and the result is again converted into a polynomial expression.

- If the first argument is an integer or a polynomial, then the result is a polynomial of domain type `DOM_POLY`; otherwise it is a polynomial expression.
- `genpoly` is a function of the system kernel.

Example 1. We create a polynomial `p` in the indeterminate `x` such that `p(7) = 15`. The coefficients of `p` are between `-3` and `3`:

```
>> p := genpoly(15, 7, x)

                        poly(2 x + 1, [x])

>> p(7)

                               15
```

Here is an example with a polynomial expression as input:

```
>> p := genpoly(15*y^2 - 6*y + 3*z, 7, x)

                                 2         2
                   y + 3 z - x y + y  + 2 x y
```

The return value has the same type as the first argument:

```
>> p := genpoly(poly(15*y^2 + 8*z, [y, z]), 7, x)

                     2           2
             poly(2 x y  + x z + y  + z, [x, y, z])
```

We check the result:

```
>> p(7, y, z)

                                  2
                          8 z + 15 y
```

`getpid` – the process ID of the running MuPAD kernel

On UNIX and Linux systems, `getpid()` returns the process ID of the running MuPAD kernel.

Call(s):

- `getpid()`

Return Value: a nonnegative integer.

Related Functions: `sysname`, `system`

Details:

- On operating systems other than UNIX or Linux, `getpid` returns `0`.
- The process ID may be useful information to communicate with other processes or to send UNIX commands to the operating system via `system`.
- `getpid` is a function of the system kernel.

Example 1. Querying the process ID of the running kernel may produce a result like this:

```
>> getpid()
```

```
16184
```

`getprop` – query properties of expressions

`getprop(f)` returns a mathematical property of the expression `f`.

Call(s):

- `getprop(f)`
- `getprop()`

Parameters:

`f` — an arithmetical expression

Return Value: `getprop(f)` returns a property of type `Type::Property`, or the expression `f` itself. The call `getprop()` returns a property or the identifier `Global`.

Related Functions: `assume`, `is`, `property::hasprop`, `Type::Property`, `unassume`

Details:

- The property mechanism helps to simplify expressions involving identifiers that carry "mathematical properties". The function `assume` allows to attach basic properties ("assumptions") such as '`x` is a real number' or '`x` is an odd integer' to an identifier `x`, say. Arithmetical expressions involving `x` may inherit such properties. E.g., '`1 + x^2` is positive' if '`x` is a real number'.

  `getprop(f)` examines the properties of all identifiers in the expression `f` and derives a property of `f`.

  See the `property` library for a description of all available properties.

- If the identifiers inside an expression have no properties, then `getprop` returns the expression itself. In particular, if `f` is an identifier without properties, then the result is again `f`. Cf. example 2.

  An exception to this rule is the case where `f` involves one of the special functions `abs`, `Re`, or `Im` with symbolic arguments. Independent of the argument, these function values always represent real numbers, which may give rise to a property of the whole expression `f`. Cf. example 4.

- The call `getprop()` returns the current "global property". See `assume` for details on setting global properties.

  The protected identifier `Global` is used to store global properties. If no global property is set, the identifier `Global` is returned. Cf. example 3.

- Only basic mathematical properties can be represented with the available properties. Therefore, `getprop` performs certain simplifications during the derivation of a property for an expression. Thus it may happen that `getprop` derives a property that is weaker than the most specific property that can be derived mathematically. Cf. example 5.

- The function `is` matches the properties of an expression with a given property.

Example 1. If `x` is an integer, then `x^2 + 1` must be a positive integer number:

```
>> assume(x, Type::Integer):
   getprop(x^2 + 1)

                  [1, infinity[ of Type::PosInt
```

If `x` represents a number in the interval `[1, infinity[`, the expression `1 - x` has the following property:

```
>> assume(x, Type::Interval([1], infinity)):
   getprop(1 - x)
```

```
                    ]-infinity, 0]
```

```
>> unassume(x):
```

Example 2. An expression is returned unchanged if it is constant, or if no properties are attached to the identifiers involved:

```
>> getprop(x), getprop(x + 2*y), getprop(sin(3))

                 x, x + 2 y, sin(3)
```

Example 3. Properties that are assumed for all identifiers are stored in the global variable `Global`. Presently, no global property is set:

```
>> getprop()

                     Global
```

In the following, a global property is set. Now, all identifiers have this property:

```
>> assume(Type::Real):
   getprop(x), getprop(y), getprop((x + y)^2 + 1/2)

    Type::Real, Type::Real, [1/2, infinity[ of Type::Positive
```

The functions `getprop` and `is` combine the global property and the properties of individual identifiers with the logical "and":

```
>> assume(Type::Positive):
   assume(x, Type::Integer):
   getprop(x)

                  Type::PosInt
```

The global property may contradict the individual properties. In this case the "empty property" `property::Null` is returned:

```
>> assume(Type::Positive):
   assume(x < 0):
   getprop(x)

                 property::Null
```

```
>> delete x: unassume():
```

Example 4. The functions `abs`, `Re`, and `Im` have a "minimal property": they produce real values. In fact, `abs` produces nonnegative real values:

```
>> delete x:
   getprop(abs(x)), getprop(Re(x)), getprop(Im(x))

            Type::NonNegative, Type::Real, Type::Real
```

Example 5. The set containing the squares of all prime numbers cannot be represented by one of the properties available in the `Type` library. Therefore, `getprop` returns the weaker property 'x^2 is a positive integer':

```
>> assume(x, Type::Prime):
   getprop(x^2)

                         Type::PosInt

>> unassume(x):
```

`ground` – ground term (constant coefficient) of a polynomial

`ground(p)` returns the constant coefficient $p(0, 0, \dots)$ of the polynomial p.

Call(s):

- `ground(p)`
- `ground(f)`
- `ground(f, vars)`

Parameters:

`p`	—	a polynomial of type `DOM_POLY`
`f`	—	a polynomial expression
`vars`	—	a list of indeterminates of the polynomial: typically, identifiers or indexed identifiers

Return Value: an element of the coefficient ring of `p`, an arithmetical expression, or `FAIL`.

Overloadable by: `p`, `f`

Related Functions: `coeff`, `collect`, `degree`, `degreevec`, `lcoeff`, `ldegree`, `lmonomial`, `lterm`, `nterms`, `nthcoeff`, `nthmonomial`, `nthterm`, `poly`, `poly2list`, `tcoeff`

Details:

- The first argument can either be a polynomial expression, or a polynomial generated by `poly`, or an element of some polynomial domain overloading `ground`.
- If the first argument `f` is not element of a polynomial domain, then `ground` converts the expression to a polynomial via `poly(f)`. If a list of indeterminates is specified, then the polynomial `poly(f, vars)` is considered.

  The constant coefficient is returned as an arithmetical expression.
- The result of `ground` is not fully evaluated. Evaluation can be enforced by the function `eval`. Cf. example 2.
- `ground` returns `FAIL` if `f` cannot be converted to a polynomial in the specified indeterminates. Cf. example 3.

Example 1. We demonstrate how the indeterminates influence the result:

```
>> f := 2*x^2 + 3*y + 1:
   ground(f), ground(f, [x]), ground(f, [y]),
   ground(poly(f)), ground(poly(f, [x])), ground(poly(f, [y]))

                               2                     2
           1, 3 y + 1, 2 x  + 1, 1, 3 y + 1, 2 x  + 1
```

The result is the evaluation at the origin:

```
>> subs(f, x = 0, y = 0), subs(f, x = 0), subs(f, y = 0)

                                         2
                      1, 3 y + 1, 2 x  + 1
```

Note the difference between `ground` and `tcoeff`:

```
>> g := 2*x^2 + 3*y:
   ground(g), ground(g, [x]);
   tcoeff(g), tcoeff(g, [x]);

                              0, 3 y

                              3, 3 y

>> delete f, g:
```

Example 2. The result of `ground` is not fully evaluated:

```
>> p := poly(27*x^2 + a, [x]): a := 5:
   ground(p), eval(ground(p))

                                    a, 5

>> delete p, a:
```

Example 3. The following expression is syntactically not a polynomial expression, and `ground` returns `FAIL`:

```
>> f := (x^2 - 1)/(x - 1): ground(f)

                                    FAIL
```

After cancellation via `normal`, `ground` can compute the constant coefficient:

```
>> ground(normal(f))

                                     1

>> delete f:
```

`has` – check if an object occurs in another object

`has(object1, object2)` checks, whether `object2` occurs syntactically in `object1`.

Call(s):

- `has(object1, object2)`
- `has(object1, l)`

Parameters:

`object1, object2`	—	arbitrary MuPAD objects
`l`	—	a list or a set

Return Value: either `TRUE` or `FALSE`

Overloadable by: `object1`

Related Functions: `_in`, `_index`, `contains`, `hastype`, `op`, `subs`, `subsex`

Details:

- `has` is a fast test for the existence of sub-objects or subexpressions. It works syntactically, i.e., mathematically equivalent objects are considered to be equal only if they are syntactically identical. See example 2.

- If `object1` is an expression, then `has(object1, object2)` tests whether `object1` contains `object2` as a subexpression. Only complete subexpressions and objects occurring in the 0th operand of a subexpression are found (see example 1).

- If `object1` is a container, then `has` checks whether `object2` occurs in an entry of `object1`. See example 4.

- If the second argument is a list or a set `l`, then `has` returns `TRUE` if at least one of the elements in `l` occurs in `object1` (see example 3). In particular, if `l` is the empty list or the empty set, then the return value is `FALSE`.

- If `object1` is an element of a domain with a `"has"` slot, then the slot routine is called with the same arguments, and its result is returned. If the domain does not have such a slot, then `FALSE` will be returned. See example 6.

  If `has` is called with a list or set as second argument, then the `"has"` slot of the domain of `object1` is called for each object of the list or the set. When the first object is found that occurs in `object1`, the evaluation is terminated and `TRUE` is returned. If none of the objects occurs in `object1`, `FALSE` will be returned.

- `has` is a function of the system kernel.

Example 1. The given expression has `x` as an operand:

```
>> has(x + y + z, x)
```

TRUE

Note that `x + y` is not a complete subexpression. Only `x`, `y`, `z` and `x + y + z` are complete subexpressions:

```
>> has(x + y + z, x + y)
```

FALSE

However, `has` also finds objects in the 0th operand of a subexpression:

```
>> has(x + sin(x), sin)
```

TRUE

Every object occurs in itself:

```
>> has(x, x)
```

TRUE

Example 2. `has` works in a purely syntactical fashion. Although the two expressions `y*(x + 1)` and `y*x + y` are mathematically equivalent, they differ syntactically:

```
>> has(sin(y*(x + 1)), y*x + y),
   has(sin(y*(x + 1)), y*(x + 1))
```

FALSE, TRUE

Complex numbers are not regarded as atomic objects:

```
>> has(2 + 5*I, 2), has(2 + 5*I, 5), has(2 + 5*I, I)
```

TRUE, TRUE, TRUE

In contrast, rational numbers are considered to be atomic:

```
>> has(2/3*x, 2), has(2/3*x, 3), has(2/3*x, 2/3)
```

FALSE, FALSE, TRUE

Example 3. If the second argument is a list or a set, `has` checks whether one of the entries occurs in the first argument:

```
>> has((x + y)*z, [x, t])
```

TRUE

0th operands of subexpressions are checked as well:

```
>> has((a + b)*c, {_plus, _mult})
```

TRUE

Example 4. `has` works for lists, sets, tables, and arrays:

```
>> has([sin(f(a) + 2), cos(x), 3], {f, g})
```

```
TRUE
```

```
>> has({a, b, c, d, e}, {a, z})
```

```
TRUE
```

```
>> has(array(1..2, 1..2, [[1, 2], [3, 4]]), 2)
```

```
TRUE
```

For an array `A`, the command `has(A,NIL)` checks whether the array has any uninitialized entries:

```
>> has(array(1..2, 1 = x), NIL),
   has(array(1..2, [2, 3]), NIL)
```

```
TRUE, FALSE
```

For tables, `has` checks indices, entries, as well as the internal operands of a table, given by equations of the form `index=entry`:

```
>> T := table(a = 1, b = 2, c = 3):
   has(T, a), has(T, 2), has(T, b = 2)
```

```
TRUE, TRUE, TRUE
```

Example 5. `has` works syntactically. Although the variable `x` does not occur mathematically in the constant polynomial `p` in the following example, the identifier `x` occurs syntactically in `p`, namely, in the second operand:

```
>> delete x: p := poly(1, [x]):
   has(p, x)
```

```
TRUE
```

Example 6. The second argument may be an arbitrary MuPAD object, even from a user-defined domain:

```
>> T := newDomain("T"):
   e := new(T, 1, 2);
   f := [e, 3];
```

```
new(T, 1, 2)

[new(T, 1, 2), 3]
```

```
>> has(f, e), has(f, new(T, 1))
```

```
TRUE, FALSE
```

If the first argument of `has` belongs to a domain without a `"has"` slot, then `has` always returns `FALSE`:

```
>> has(e, 1)
```

```
FALSE
```

Users can overload `has` for their own domains. For illustration, we supply the domain `T` with a `"has"` slot, which puts the internal operands of its first argument in a list and calls `has` for the list:

```
>> T::has := (object1, object2) -> has([extop(object1)], object2):
```

If we now call `has` with the object `e` of domain type `T`, the slot routine `T::has` is invoked:

```
>> has(e, 1), has(e, 3)
```

```
TRUE, FALSE
```

The slot routine is also called if an object of domain type `T` occurs syntactically in the first argument:

```
>> has(f, 1), has(f, 3)
```

```
TRUE, TRUE
```

`hastype` – test if an object of a specified type occurs in another object

`hastype(object, T)` tests if an object of type `T` occurs syntactically in `object`.

Call(s):

- `hastype(object, T <, inspect>)`

Parameters:

- `object` — an arbitrary MuPAD object
- `T` — a type specifier, or a set or a list of type specifiers
- `inspect` — a set of domain types

Return Value: either TRUE or FALSE.

Overloadable by: `object`

Related Functions: `domtype`, `has`, `misc::maprec`, `testtype`, `Type`, `type`

Details:

- `hastype(object, T)` tests if a sub-object `s` of type `T` occurs in `object`, i.e., such that `testtype(s, T)` returns `TRUE`.
- The type specifier `T` may be either a domain type such as `DOM_INT`, `DOM_EXPR` etc., a string as returned by the function `type`, or a `Type` object. The latter are probably the most useful predefined values for the argument `T`.

  If `T` is not a valid type specifier, then `hastype` returns `FALSE`.

  See example 1.
- If `object` is an expression, then `hastype(object, T)` tests whether `object` contains a subexpression of type `T`; see example 1.

  If `object` is a container, then `hastype` checks whether a sub-object of type `T` occurs in an entry of `object`; see example 3.
- If the second argument is a list or a set, `hastype` checks whether a sub-object of one of the types in `T` occurs in `object`. Cf. example 1.
- `hastype` works in a recursive fashion and descends into the following objects: expressions, arrays, lists, sets, and tables; see example 3.

  `hastype` does not step into the other basic domains, such as rational numbers, complex numbers, polynomials, or procedures; see example 2.
- If the third argument `inspect` is present, then `hastype` also steps recursively into sub-objects of the domain types given in `inspect`; cf. example 2.
- `hastype` looks only for sub-objects that are syntactically of type `T`. Properties of identifiers set via `assume` are not taken into account; cf. example 3.

  NOTE

Example 1. In this example, we first test if a given expression has a subexpression of type `DOM_FLOAT`:

```
>> hastype(1.0 + x, DOM_FLOAT)

                         TRUE

>> hastype(1 + x, DOM_FLOAT)
```

FALSE

We may also test if an expressions contains a subexpression of one of the two types `DOM_FLOAT` or `DOM_INT`:

```
>> hastype(1.0 + x, {DOM_FLOAT, DOM_INT})
```

TRUE

While the first of following two tests returns `FALSE`, since `tan` is not a valid type specifier, the second test yields `TRUE`, since the given expression contains a subexpression of type `"tan"`:

```
>> hastype(sin(tan(x) + 1/exp(1 - x)), tan),
   hastype(sin(tan(x) + 1/exp(1 - x)), "tan")
```

FALSE, TRUE

You can also use type specifiers from the `Type` library:

```
>> hastype([-1, 10, -5, 2*I], Type::PosInt)
```

TRUE

Example 2. We demonstrate the use of the optional third argument. We want to check if a procedure contains a subexpression of type `"float"`. By default, `hastype` does not descend recursively into a procedure:

```
>> f := x -> float(x) + 3.0:
   hastype(f, "float")
```

FALSE

You can use the third argument to request the inspection of procedures explicitly:

```
>> hastype(f, "float", {DOM_PROC})
```

TRUE

Also, by default, `hastype` does not descend recursively into the basic domains `DOM_COMPLEX` and `DOM_RAT`:

```
>> hastype(1 + I, DOM_INT), hastype(2/3,  DOM_INT)
```

FALSE, FALSE

In order to inspect these data types, one has to use the third argument:

```
>> hastype(1 + I, DOM_INT, {DOM_COMPLEX}),
   hastype(2/3,  DOM_INT, {DOM_RAT})
```

TRUE, TRUE

It is also possible to inspect domains elements using the third argument. As an example let us define a matrix element and ask for a subexpression of type integer:

```
>> A:=matrix([[1, 1], [1, 0]]):
   hastype(A, DOM_INT),  hastype(A, DOM_INT, {Dom::Matrix()})
```

FALSE, TRUE

Example 3. We demonstrate how `hastype` effects on container objects. Let us first stress tables:

```
>> hastype(table(1 = a), DOM_INT), hastype(table(a = 1), DOM_INT)
```

FALSE, TRUE

As shown, `hastype` does not inspect the indices of a table, but checks recursively whether a sub-object of a given type occurs in an entry. This is also true for arrays, lists and sets:

```
>> hastype(array(1..4, [1, 2, 3, 4]), DOM_INT),
   hastype([1, 2, 3, 4], DOM_INT),
   hastype({1, 2, 3, 4}, DOM_INT),
   hastype([[a, [1]], b, c], DOM_INT)
```

TRUE, TRUE, TRUE, TRUE

`hastype` can only work syntactically, i.e. properties are not taken into account:

```
>> assume(a,Type::Integer):
   hastype([a, b], Type::Integer), hastype([a, b], DOM_INT)
```

FALSE, FALSE

```
>> delete a:
```

`heaviside` – the Heaviside step function

`heaviside(x)` represents the Heaviside step function.

Call(s):

- `heaviside(x)`

Parameters:

`x` — an arithmetical expression

Return Value: an arithmetical expression.

Overloadable by: `x`

Related Functions: `dirac`

Details:

- If the argument represents a positive real number, then 1 is returned. If the argument represents a negative real number, then 0 is returned. If the argument is a complex number of domain type `DOM_COMPLEX`, then `undefined` is returned. For all other arguments, an unevaluated function call is returned.
- `heaviside` does not have a predefined value at the origin. Use

  `sysassign(heaviside(0),myValue)`

  and

  `sysassign(heaviside(float(0)),myFloatValue)`

  to assign your favorite values.
- The derivative of `heaviside` is the delta distribution `dirac`.

Example 1. `heaviside` returns 1 or 0 for arguments representing positive or negative real numbers, respectively:

```
>> heaviside(-3), heaviside(-sqrt(3)), heaviside(-2.1),
   heaviside(PI - exp(1)), heaviside(sqrt(3))

                        0, 0, 0, 1, 1
```

Arguments of domain type `DOM_COMPLEX` yield `undefined`:

```
>> heaviside(1 + I), heaviside(2/3 + 7*I), heaviside(0.1*I)

                undefined, undefined, undefined
```

An unevaluated call is returned for other arguments:

```
>> heaviside(0), heaviside(x), heaviside(ln(-5)), heaviside(x + I)
```

```
heaviside(0), heaviside(x), heaviside(I PI + ln(5)),

   heaviside(x + I)
```

Natural values at the origin are 0, 1/2, or 1:

```
>> prev_protection:= unprotect(heaviside):
   heaviside(0) := 1/2: heaviside(0)

                                1/2
```

```
>> delete heaviside(0):
   protect(heaviside, prev_protection):
   delete prev_protection:
```

```
>> heaviside(0)

                            heaviside(0)
```

Example 2. `heaviside` reacts to assumptions set by `assume`:

```
>> assume(x > 0): heaviside(x)

                                 1
```

```
>> unassume(x):
```

Example 3. The derivative of `heaviside` is the delta distribution `dirac`:

```
>> diff(heaviside(x - 4), x)

                            dirac(x - 4)
```

The integrator `int` handles `heaviside`:

```
>> int(exp(-x)*heaviside(x), x = -infinity..infinity)

                                 1
```

We do not recommend to use `heaviside` in numerical integration. It is much more efficient to split the quadrature into pieces, each of which having a smooth integrand:

```
>> DIGITS := 3: numeric::int(exp(-x)*heaviside(x^2 - 2), x=-3..10)

                                16.2
```

```
>> numeric::int(exp(-x), x = -3..-2^(1/2)) +
   numeric::int(exp(-x), x = 2^(1/2)..10)

                              16.2

>> delete DIGITS:
```

---

help – display a help page

help("word") or ?word displays the online help page related to word.

Call(s):

- help("word")
- ?word

Parameters:

word — any keyword

Return Value: the void object null() of type DOM_NULL.

Related Functions: info, Pref::ansi

Details:

- help("word") displays a help page with information about the keyword "word".
- The exact form of the output depends on the platform. In the terminal version, information is displayed as ASCII text. In XMuPAD and on MacOS or Windows platforms, the help page is displayed in a separate help window with hypertext functionality. Highlighted words are hypertext links, which can be followed by clicking on them with the mouse.
- The call ?word is a short form of help("word"). The ? command is not a MuPAD function and cannot be used in expressions. It can only be entered interactively and in a line of its own. The syntax ?word may not be used in files to be read. Note that the search word word must neither be put in quotation marks nor followed by a terminating semicolon.
- The keyword may contain the wildcards ? and *. ? represents any single character, and * represents an arbitrarily long, possibly empty, sequence of characters. There are three exceptions: ?* and ?`*` lead directly to the help page for _mult, and ?`**` leads to the help page for _power. Cf. example 1.

- If there is no help page for the specified keyword, then a list of keywords with similar spelling is displayed. Cf. example 2.
- The command `anames(All)` returns a set with the names of all currently loaded system functions. The command `?**` returns a list of all available help pages.
- The documentation for modules is not included in this help system. Example 3 demonstrates the calls for obtaining information about modules and their functions.
- `help` is a function of the system kernel.

Example 1. `help` expands wildcards:

```
>> ?*type

 Try:  ReturnType domtype hastype testtype type Type
  Type::AnyType
```

An exception: `?*` leads directly to the help page for `_mult`:

```
>> ?*

 * -- multiply expressions

 Introduction

  a * b respectively _mult(a, b) computes the product a*b.

 Call(s)

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

 Parameters

  a, b  -  arithmetical expressions

 [...]
```

Example 2. There is no information on the non-existent function `worm`:

```
>> ?worm

 Sorry, no help page available for 'worm' !

 Try:  Word norm
```

Example 3. MuPAD supports `C++` compiled kernel extensions, called dynamic modules. The documentation of a dynamic module is not integrated into the MuPAD hypertext help system, but is provided as plain text online documentation, which can be displayed via the `"doc"` method of the corresponding module, e.g., `util::doc`:

```
>> module(util): util::doc()

 MODULE
   util - A collection of utility functions

 INTRODUCTION
   The module provides a collection of useful utility functions.

 INTERFACE
   util::busyWaiting, util::date,      util::doc,
   util::kernelPath,  util::kernelPid, util::sleep,
   util::time,        util::userName

>> util::doc("kernelPath")

 NAME
   util::kernelPath - Returns the pathname of the MuPAD kernel

 SYNOPSIS
   util::kernelPath()

 DESCRIPTION
   This function returns the pathname of the MuPAD kernel.

 EXAMPLES
   >> util::kernelPath()

      "C:\\PROGRA~1\\SCIFACE\\MUPADP~1.5\\BIN\\MUPKERN.EXE"

   >> util::kernelPath()

      "/usr/local/mupad/linux/bin/mupad"

 SEE ALSO
   util::kernelPid, util::userName
```

Background:

- In the terminal version, the viewer called to display the help pages in ASCII format is given by the system variable `PAGER`. See `Pref::ansi` on how to control the format of this output.

`history` – access an entry of the history table

`history(n)` returns the `n`th entry of the history table.

`history()` returns the index of the most recent entry in the history table.

Call(s):

- `history(n)`
- `history()`

Parameters:

`n` — a positive integer

Return Value: `history(n)` returns a list with two elements, and `history()` returns a nonnegative integer.

Related Functions: `fread`, `HISTORY`, `last`, `read`

Details:

- The commands that are entered interactively in a MuPAD session, executed in a procedure, or read from a file, as well as the resulting MuPAD outputs are stored in an internal data structure, the history table. `history()` returns the index of the most recent entry in the history table. At interactive level, this is the number of commands that have been entered since the start of the session or the last restart.
- `history(n)` returns the `n`th entry in the history table in form of a list with two elements. The first element of this list is a MuPAD command, and the second element is the result of this command returned by MuPAD. The order of the entries in the history table is such that larger indices correspond to more recent entries.
- The command `last` accesses the result entries from the history table. The call `last(n)` is equivalent to `history(history() - n + 1)[2]` at interactive level.
- The environment variable `HISTORY` determines the maximal number of history entries that are stored at interactive level. The default value is `20`. Only the most recent entries are kept in memory. Thus valid arguments for `history` are all integers between `history() - HISTORY + 1` and `history()`. All other integers lead to an error message.

- The result returned by `history` is not evaluated again (see example 2). Use the function `eval` to force a subsequent evaluation.
- Commands and their results are stored in the history table even if the output is suppressed by a colon. See example 1.
- Compound statements, such as `for`, `repeat`, and `while` loops, `if` and `case` branching instructions, and procedure definitions via `proc` are stored in the history table as a whole at interactive level. See the help page of `last` for examples.
- Commands appearing on the same input line lead to separate entries in the history table if they are separated by a colon or a semicolon. In contrast, a statement sequence is regarded as a single command (see example 3).
- Commands that are read from a file via `fread` or `read` are stored in the history table, and at last the `fread` or `read` command is stored in the history table (because the `fread` or `read` command is finished foremost after reading the file). However, if the option *Plain* is used, then a separate history table is in effect within the file, and the commands from the file do not appear in the history table of the enclosing context.
- Note that every call of `history` modifies the history table and possibly erases the earliest history entry.
- Every procedure has its own local history table. However, the entries of this table cannot be accessed via `history` (see `last`); the command `history` always refers to the history table at interactive level.
- `history` is a function of the system kernel.

Example 1. The index of the most recent entry in the history table increases by one for each entered command, also by `history()`. Note that every command is stored in the history table, whether its output is suppressed by a colon or not:

```
>> history(); sqrt(1764); history(): history()

                                    3

                                    42

                                    6
```

`history(history())` returns a list with two elements. The first element is the last command, and the second element is the result returned by MuPAD, which is equal to `last(1)` or `%`:

```
>> int(2*x*exp(x^2), x);
   history(history()), last(1)
```

```
                          2
                    exp(x )

             2             2          2
[int(2 x exp(x ), x), exp(x )], exp(x )
```

The following command returns the next to last command and its result:

```
>> history(history() - 1)

                     2             2
        [int(2 x exp(x ), x), exp(x )]
```

A restart cleans up the history table:

```
>> reset():
   history()

                                4
```

The output of the command `history()` above depends on the number of commands in your MuPAD startup file `userinit.mu`.

Example 2. First `a` should be `0`:

```
>> a := 0:
   a

                                0
```

Now `1` is assigned to `a`:

```
>> a := 1:
   a

                                1
```

The command `history(history()-2)` refers to the command `a` after assigning `0` to `a`, the return value of `history` is not the new value of `a`, because the result returned by `history` is not evaluated again:

```
>> history(history() - 2)

                              [a, 0]
```

Example 3. The both commands leads to two entries in the history table. The command `history(history()-1)` returns only the last command `b:=a`, not both commands:

```
>> a := 0: b := a:
   history(history() - 1)
```

```
                    [(a := 0), 0]
```

If the commands are entered as an statement sequence (enclosed in `()`), they leads to one entry. `history(history())` picks out the last command, that is the statement sequence:

```
>> (a := 0; b := a;):
   history(history())
```

```
                    [(a := 0;
                    b := a), 0]
```

The last input

```
>> type(op(%, 1))
```

```
                    "_stmtseq"
```

`hold` – delay evaluation

`hold(object)` prevents the evaluation of `object`.

Call(s):

- `hold(object)`

Parameters:

`object` — any MuPAD object

Return Value: the unevaluated object.

Further Documentation: Chapter 5 of the MuPAD Tutorial.

Related Functions: `context`, `delete`, `eval`, `freeze`, `genident`, `indexval`, `level`, `proc`, `val`

Details:

- When a MuPAD object is entered interactively, then the system evaluates it and returns the evaluated result. When a MuPAD object is passed as an argument to a procedure, then the procedure usually evaluates the argument before processing it. *Evaluation* means that identifiers are replaced by their values and function calls are executed. `hold` is intended to prevent such an evaluation when it is undesirable.

- A typical application of `hold` is when a function that can only process numerical arguments, but not symbolical ones, is to be plotted or to be subjected to some numerical algorithm. See example 6.

- Another possible reason for using `hold` is efficiency. For example, if a function call `f(x, y)` with symbolic arguments is passed as argument to another function, but is known to return itself symbolically, then the possibly costly evaluation of the "inner" function call can be avoided by passing the expression `hold(f)(x, y)` as argument to the "outer" function instead. Then the arguments `x, y` are evaluated, but the call to `f` is not executed. See examples 1 and 7.

- Since using `hold` may lead to strange effects, it is recommended to use it only when absolutely necessary.

- `hold` only delays the evaluation of an object, but cannot completely prevent it on the long run; see example 5.

  You can use `freeze` to completely prevent the evaluation of a procedure or a function environment.

- A MuPAD procedure can be declared with the option `hold`. This has the effect that arguments are passed to the procedure unevaluatedly. See the help page of `proc` for details.

- The functions `eval` or `level` can be used to force a subsequent evaluation of an unevaluated object (see example 2). In procedures with `option hold`, use `context` instead.

- `hold` is a function of the system kernel.

Example 1. In the following two examples, the evaluation of a MuPAD expression is prevented using `hold`:

```
>> x := 2:
   hold(3*0 - 1 + 2^2 + x)

                        2
             3 0 - 1 + 2  + x
```

```
>> hold(error("not really an error"))
```

```
error("not really an error")
```

Without `hold`, the results would be as follows:

```
>> x := 2:
   3*0 - 1 + 2^2 + x
```

```
5
```

```
>> error("not really an error")
```

```
Error: not really an error
```

The following command prevents the evaluation of the operation `_plus`, but not the evaluation of the operands:

```
>> hold(_plus)(3*0, -1, 2^2, x)
```

```
0 - 1 + 4 + 2
```

Note that in the preceding example, the arguments of the function call are evaluated, because `hold` is applied only to the function `_plus`. In the following example, the argument of the function call is evaluated, despite that fact that `f` has the option hold:

```
>> f := proc(a)
          option hold;
        begin
          return(a + 1)
        end_proc:
   x := 2:
   hold(f)(x)
```

```
f(2)
```

This happens for the following reason. When `f` is evaluated, the option `hold` prevents the evaluation of the argument `x` of `f` (see the next example). However, if the evaluation of `f` is prevented by `hold`, then the option `hold` has no effect, and MuPAD evaluates the operands, but not the function call.

The following example shows the expected behavior:

```
>> f(x), hold(f(x))
```

```
x + 1, f(x)
```

The function `eval` undoes the effect of `hold`. Note that it yields quite different results, depending on how it is applied:

```
>> eval(f(x)), eval(hold(f)(x)), eval(hold(f(x))), eval(hold(f))(x)
```

```
3, 3, x + 1, x + 1
```

Example 2. Several `hold` calls can be nested to prevent subsequent evaluations:

```
>> x := 2:
   hold(x), hold(hold(x))
```

```
                    x, hold(x)
```

The result of `hold(hold(x))` is the unevaluated operand of the outer call of `hold`, that is, `hold(x)`. Applying `eval` evaluates the result `hold(x)` and yields the unevaluated identifier `x`:

```
>> eval(%)
```

```
                      2, x
```

Another application of `eval` yields the value of `x`:

```
>> eval(%)
```

```
                      2, 2
```

```
>> delete x, f:
```

Example 3. The following command prevents the evaluation of the operation `_plus`, replaces it by the operation `_mult`, and then evaluates the result:

```
>> eval(subsop(hold(_plus)(2, 3), 0 = _mult))
```

```
                        6
```

Example 4. The function `domtype` evaluates its arguments:

```
>> x := 0:
   domtype(x), domtype(sin), domtype(x + 2)
```

```
          DOM_INT, DOM_FUNC_ENV, DOM_INT
```

Using `hold`, the domain type of the unevaluated objects can be determined: `x` and `sin` are identifiers, and `x + 2` is an expression:

```
>> domtype(hold(x)), domtype(hold(sin)), domtype(hold(x + 2))
```

```
          DOM_IDENT, DOM_IDENT, DOM_EXPR
```

Example 5. `hold` prevents only one evaluation of an object, but it does not prevent evaluation at a later time. Thus using `hold` to obtain a a symbol without a value is usually not a good idea:

```
>> x := 2:
   y := hold(x);
   y
```

```
                              x

                              2
```

In this example, deleting the value of the identifier `x` makes it a symbol, and using `hold` is not necessary:

```
>> delete x:
   y := x;
   y
```

```
                              x

                              x
```

However, the best way to obtain a new symbol without a value is to use `genident`:

```
>> y := genident("x");
   y
```

```
                             x1

                             x1
```

```
>> delete y:
```

Example 6. Suppose that we want to plot the graph of the piecewise continuous function $f(x)$ that is identically zero on the negative real axis and equal to $\exp(-x)$ on the positive real axis:

```
>> f := x -> if x < 0 then 0 else exp(-x) end_if:
```

If we pass the symbolical expression `f(x)` as an argument to `plotfunc2d`, then an error occurs:

```
>> delete x:
   plotfunc(f(x), x = -2..2)
```

```
 Error: Can't evaluate to boolean [_less];
 during evaluation of 'f'
```

The reason is that `plotfunc2d` evaluates its arguments, and the evaluation of `f(x)` for a symbolical argument `x` leads to an error:

```
>> f(x)

 Error: Can't evaluate to boolean [_less];
 during evaluation of 'f'
```

A solution is to use `hold`:

```
>> plotfunc2d(hold(f)(x), x = -2..2):
```

The same phenomenon occurs when we want to apply numerical integration to `f`:

```
>> numeric::int(f(x), x = -2..2)

 Error: Can't evaluate to boolean [_less];
 during evaluation of 'f'

>> numeric::int(hold(f)(x), x = -2..2)

                           0.8646647168
```

Example 7. The function `int` is unable to compute a closed form of the following integral and returns a symbolic `int` call:

```
>> int(sin(x)*sqrt(sin(x) + 1), x)

                                       1/2
                 int(sin(x) (sin(x) + 1)   , x)
```

After the change of variables `sin(x)=t`, a closed form can be computed:

```
>> t := time():
   f := intlib::changevar(int(sin(x)*sqrt(sin(x) + 1), x), sin(x) = y);
   time() - t;
   eval(f)

                        /          1/2    \
                        | y (y + 1)       |
                     int| ------------, y |
                        |         2 1/2   |
                        \   (1 - y )      /

                              9210

                         1/2 / 2 y       \
                 (y - 1) (y + 1)    | --- + 4/3 |
                                    \  3        /
                 --------------------------------
                                 2 1/2
                            (1 - y )
```

Measuring computing times with `time` shows: Most of the time in the call to `intlib::changevar` is spent in re-evaluating the argument. This can be prevented by using `hold`:

```
>> t := time():
   f := intlib::changevar(hold(int)(sin(x)*sqrt(sin(x) + 1), x),
                          sin(x) = y);
   time() - t;

                           /          1/2    \
                           | y (y + 1)       |
                        int| ------------, y |
                           |        2 1/2    |
                           \  (1 - y )       /

                                  20
```

..., hull – convert to a floating point interval

`hull(obj)` returns a floating point interval enclosing `obj`.

Call(s):

- `l ... r`
- `hull(object)`

Parameters:

`l, r, object` — arbitrary MuPAD objects

Return Value: a floating point interval, the empty set, or `FAIL`.

Overloadable by: `object`

Side Effects: The function is sensitive to the environment variable `DIGITS` which determines the numerical working precision. Each sub-object of `object` may be evaluated multiple times and should not have any side-effects.

Related Functions: `Dom::FloatIV`, `DIGITS`, `float`, `interval`, `Pref::floatFormat`, `Pref::trailingZeroes`

Details:

- `l ... r` is equivalent to `hull(l, r)`.
- `hull` converts numerical and interval expressions to numerical intervals of type `DOM_INTERVAL`. It accepts lists and sets of numerical expressions or intervals as well as numerical expressions, intervals, and set-theoretic functions of intervals and sets.
- `...`, `hull` is mapped recursively to the operands of any expression given—but for subexpressions, lists and sets are not accepted. Identifiers are replaced by intervals, respecting a certain subset of properties. Cf. example 3. Likewise, function calls and domain elements not overloading `...`, `hull` are converted to the interval representing the complex plane.
- The output of floating point intervals is influenced by the same parameters as the output of floating point numbers:

  `DIGITS`, `Pref::floatFormat`, and `Pref::trailingZeroes`.

Example 1. `hull` returns an interval enclosing its arguments. You can also use the operator `...` instead of the function call:

```
>> hull(0, PI) = 0 ... PI

            0.0 ... 3.141592654 = 0.0 ... 3.141592654
```

Infinities are displayed using `RD_NINF` for $-\infty$ and `RD_INF` for ∞:

```
>> hull(-infinity, 1/4, 9/7), hull({-infinity, 1/4, 9/7})

        RD_NINF ... 1.285714286, RD_NINF ... 1.285714286
```

Please note that any number whose absolute value is larger than MuPAD can store in a float is considered infinite:

```
>> hull(0, 1e1000000)^4

                          0.0 ... RD_INF
```

Example 2. Inversion of intervals may lead to unions of intervals. If these are not required, you may use `hull` to unify them:

```
>> 1/(-1 ... 1); hull(%)

             RD_NINF ... -1.0 union 1.0 ... RD_INF

                      RD_NINF ... RD_INF
```

Example 3. The application of `...`, `hull` to an identifier without a value returns an interval representing the complex plane:

```
>> delete x: hull(x)

          (RD_NINF ... RD_INF) + (RD_NINF ... RD_INF) I
```

Certain properties are respected during this conversion:

```
>> assume(x > 0): hull(x);
   delete x:

                        0.0 ... RD_INF
```

This way, you can enclose the values of an expression:

```
>> hull(sin(abs(x)))

                         -1.0 ... 1.0
```

Calls to "unknown" functions are regarded as potentially returning the complex plane:

```
>> hull(f(x))

          (RD_NINF ... RD_INF) + (RD_NINF ... RD_INF) I
```

Changes:

- `...`, `hull` is a new function.

`hypergeom` – the hypergeometric functions

`hypergeom([a1, a2, ...], [b1, b2, ...], z)` represents the hypergeometric function. With $a = [a_1, a_2, \ldots, a_p]$ and $b = [b_1, b_2, \ldots, b_q]$, the hypergeometric function of order p, q is defined as

$$ {}_pF_q(a; b; z) = \sum_{k=0}^{\infty} \frac{(a_1)_k (a_2)_k \cdots (a_p)_k}{(b_1)_k (b_2)_k \cdots (b_q)_k} \, \frac{z^k}{k!}, $$

where $(c)_k = c \cdot (c+1) \cdots (c+k-1)$, $(c)_0 = 1$ is the usual Pochhammer symbol. The quantities a and b are called 'the lists for the upper and lower parameters', respectively.

Call(s):

- `hypergeom([a1, a2, ...], [b1, b2, ...], z)`

Parameters:

`a1, a2, ...`	—	the 'upper parameters': arithmetical expressions
`b1, b2, ...`	—	the 'lower parameters': arithmetical expressions
`z`	—	the 'argument': an arithmetical expression

Return Value: an arithmetical expression.

Overloadable by: `z`

Side Effects: When called with floating point arguments, these functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Details:

- The hypergeometric function is defined for complex arguments a_i, b_j, and z.

- A floating point value is returned if at least one of the arguments is a floating point number and all other arguments can be converted to floating point numbers.

- For most exact arguments, the hypergeometric function returns a symbolic function call. If an upper parameter coincides with a lower parameter, these values cancel and are removed from the parameter lists.

  The following special values are implemented:

  - ${}_pF_p(a; a; z) = {}_0F_0([\,]; [\,]; z) = e^z$
  - ${}_pF_q(a; b; z) = 1$ if the list of upper parameters `a` contains more zeroes than the list of lower parameters `b`.
  - ${}_pF_q(a; b; 0) = 1$.

  If, after cancellation of identical parameters, the upper parameters contain a negative integer larger than the largest negative integer in the lower parameters, then ${}_pF_q(a; b; z)$ is a polynomial in `z`. If all upper and lower parameters as well as the argument `z` do not contain any symbolic identifiers, a corresponding explicit result is returned. If the parameters or `z` contain symbols, expansion to the polynomial representation is available via `simplify`. Cf. example 2.

- Also empty lists $a = [\,]$ or $b = [\,]$ may be passed to `hypergeom`. The corresponding functions are:

$${}_0F_q([\,]; b; z) = \sum_{k=0}^{\infty} \frac{1}{(b_1)_k (b_2)_k \cdots (b_q)_k} \frac{z^k}{k!},$$

$$
{}_pF_0(a;[\,];z) = \sum_{k=0}^{\infty} (a_1)_k (a_2)_k \cdots (a_p)_k \,\frac{z^k}{k!},
$$

$$
{}_0F_0([\,];[\,];z) = \sum_{k=0}^{\infty} \frac{z^k}{k!} = e^z.
$$

Example 1. Symbolic calls are returned for exact or symbolic arguments:

```
>> hypergeom([], [2], x),
   hypergeom([1], [2, 3], PI),
   hypergeom([1, 1/2], [1/3], x + 3*I)

 hypergeom([], [2], x), hypergeom([1], [2, 3], PI),

    hypergeom([1/2, 1], [1/3], x + 3 I)
```

Floating point values are returned for floating point arguments:

```
>> hypergeom([], [2], 3.0),
   hypergeom([1], [2.0], PI),
   hypergeom([PI], [2, 3], 4.0),
   hypergeom([1, 2], [3, 4, 5, 6], 1.0*I),
   hypergeom([1 + I], [1/(2 + I)], 1.0*I)

 3.468649619, 7.047601352, 5.152314068,

    0.9999801588 + 0.005555508314 I,

    - 0.7438410785 - 0.5956994573 I
```

Example 2. ${}_0F_0([\,];[\,];z) = e^z$:

```
>> hypergeom([], [], z)

                    exp(z)
```

Because identical values in a and b cancel, the same is true for ${}_pF_p(a;a;z)$:

```
>> hypergeom([a, b], [a, b], z)

                    exp(z)
```

Any hypergeometric function, evaluated at 0, has the value 1:

```
>> hypergeom([a, b], [c, d, e], 0)
```

```
                                   1
```

If, after cancelling identical parameters, the list of upper parameters contains a zero, the resulting hypergeometric function is constant with the value 1:

```
>> hypergeom([0, 0, 2, 3], [a, 0, 4], z)

                                   1
```

If, after cancelling identical parameters, the upper parameters contain a negative integer larger than the largest negative integer in the lower parameters, the hypergeometric function is a polynomial. If all parameters as well as the argument z are numerical, a corresponding explicit value is returned:

```
>> hypergeom([-4, -2 , 3], [-3, 1, 4], PI*sqrt(2))

                         2
                     6 PI         1/2
                     ----- - 2 PI 2    + 1
                       5
```

For symbolic parameters or symbolic z, the polynomial representation may be obtained via simplify:

```
>> hypergeom([-40, -5], [1, 4], z) =
   simplify(hypergeom([-40, -5], [1, 4], z))

 hypergeom([-40, -5], [1, 4], z) =

                           3          4          5
                 2   2470 z    45695 z    27417 z
    50 z + 390 z  + ------- + -------- + -------- + 1
                        3         84         280

>> hypergeom([-3, a], [b], z) =
   simplify(hypergeom([-3, a], [b], z))

                                              2
                                         3 a z  (- 2 a - 2)
 hypergeom([a, -3], [b], z) = 1 - ------------------ -
                                             2 b (b + 1)

        3
    a z  (- a - 2) (- 2 a - 2)   3 a z
    -------------------------- - -----
        2 b (b + 1) (b + 2)        b
```

If the largest negative integer in the list of lower parameters is larger than the largest negative integer in the list of upper parameters, the corresponding hypergeometric function is not defined (because its definition involves a division by zero):

```
>> hypergeom([-40, -5, 3], [-3, 1, 4], z)

 Error: illegal parameters [hypergeom]
```

Example 3. The functions `float`, `diff`, and `series` handle expressions involving the hypergeometric functions:

```
>> float(ln(3 + hypergeom([17], [exp(1), ln(5)], sqrt(PI))))

                           3.488880173

>> diff(hypergeom([a, b], [c, d], x), x)

        a b hypergeom([a + 1, b + 1], [c + 1, d + 1], x)
        ------------------------------------------------
                               c d
```

Note that differentiation of a hypergeometric function w.r.t. one of its uppper or lower parameters does not, in general, lead to hypergeometric functions. Certain peculiar cases are an exception:

```
>> diff(hypergeom([a + 1, b], [a + 2], x), a)

     b x hypergeom([a + 2, a + 2, b + 1], [a + 3, a + 3], x)
     -------------------------------------------------------
                                     2
                              (a + 2)

>> series(hypergeom([1, 2], [3], x), x)

                         2      3    4      5
                  2 x    x    2 x    x    2 x       6
             1 + --- + -- + ---- + -- + ---- + O(x )
                   3    2     5     3     7
```

Expansions about ∞ are possible:

```
>> series(hypergeom([1/2], [1/3], x), x = infinity, 3)

    1/6   1/2          1/2       1/2         1/2
 2 x    PI    exp(x) 3        PI    exp(x) 3
 ------------------------ - ------------------ -
       3 gamma(2/3)                5/6
                                18 x    gamma(2/3)


        1/2          1/2
     5 PI    exp(x) 3        /   1   \
    -------------------- + O| ----- |
        11/6                 |  17/6 |
    144 x     gamma(2/3)     \ x     /
```

However, there are very few (if any) complete expansions for hypergeometric functions about any of its upper or lower parameters.

Example 4. Often, at particular choices of parameters, the hypergeometric function reduces to simpler special functions. For example, in the case of ${}_1F_1$, also known as the standard confluent hypergeometric function, the hypergeometric function can be reduced to a Bessel function if its (single) lower parameter is exactly twice its (single) upper parameter. This is verified numerically below:

```
>> v:= 1.0 + I: z:= float(PI):
   hypergeom([v + 1/2], [2*v + 1], 2*I*z) =
   (gamma(1 + v)*exp(I*z)*((z/2)^(-v))*besselJ(v, z))

 - 0.2766083174 - 0.2537119431 I =

    - 0.2766083174 - 0.2537119431 I

>> delete v, z:
```

In the following example, ${}_2F_1$, which is known as the Gauss hypergeometric function, can be reduced into a simple elementary function involving logarithms when the parameters are `[1, 1], [2]`, as verified numerically below:

```
>> eq := hypergeom([1, 1], [2], z) = -ln(1 - z)/z:
   float(subs(eq, z = 1/3)), float(subs(eq, z = 1/2))

      1.216395324 = 1.216395324, 1.386294361 = 1.386294361

>> delete eq:
```

Example 5. The interval $[1, \infty)$ is a branch cut for the hypergeometric function; the sign of the imaginary part changes when crossing the cut. The branch cut belongs to the lower branch:

```
>> eq := hypergeom([1, 1], [2], z) = -ln(1 - z)/z:
   float(subs(eq, z = 2 + I*10^(-DIGITS)))

 7.853981633e-11 + 1.570796327 I =

    7.853981634e-11 + 1.570796327 I

>> float(subs(eq, z = 2 - I*10^(-DIGITS)))

 7.853981633e-11 - 1.570796327 I =

    7.853981634e-11 - 1.570796327 I

>> float(subs(eq, z = 2))

        1.532829442e-65 - 1.570796327 I = -1.570796327 I
```

Background:

- When no b_j in the list b lies in the set $\{0, -1, -2, \dots\}$, the series

$$ {}_pF_q(a; b; z) = \sum_{k=0}^{\infty} \frac{(a_1)_k (a_2)_k \cdots (a_p)_k}{(b_1)_k (b_2)_k \cdots (b_q)_k} \frac{z^k}{k!} $$

  converges if one of the following conditions hold:

  1. $p \le q, |z| < \infty$;
  2. $p = q + 1, |z| < 1$;
  3. $p = q + 1, |z| = 1, \Re(\psi_q) > 0$;
  4. $p = q + 1, |z| = 1, z \ne 1, -1 < \Re(\psi_q) \le 0$;
  5. a contains a zero or a negative integer;

  where $\psi_q = \sum_{k=1}^{q} b_k - \sum_{j=1}^{q+1} a_j$. The series diverges in the remaining cases. If one of the parameters in a is equal to zero or a negative integer, then the series terminates, turning into what is called a hypergeometric polynomial.

  The generalized hypergeometric function of order (p, q) is given by the series definition in the region of convergence, while for $p = q + 1, |z| \ge 1$, it is defined as an analytic continuation of this series.

- The function ${}_pF_q(a; b; z)$ is symmetric w.r.t. the parameters, i.e., it does not depend on the order of the arrangement $a_1, a_2, \dots$ in a or $b_1, b_2, \dots$ in b.

- As mentioned above, if some upper parameter is equal to $n = 0, -1, -2, \dots$, the function turns into a polynomial of degree n. If we relax the condition stated above for the lower parameters b and there is some lower parameter equal to $m = 0, -1, -2, \dots$, the function ${}_pF_q(a; b; z)$ also reduces to a polynomial in z provided $n > m$. It is undefined if $m > n$ or if no upper parameter is a non-positive integer (resulting in division by zero in one of the series coefficients). The case $m = n$ is handled by the following rule.

- If for r values of the upper parameters, there are r values of the lower parameters equal to them (i.e., $a = [a_1, ..., a_{p-r}, c_1, ..., c_r]$, $b = [b_1, ..., b_{q-r}, c_1, ..., c_r]$), then the order (p, q) of the function ${}_pF_q(a; b; z)$ is reduced to $(p - r, q - r)$:

$$ {}_pF_q([a_1, \dots, c_r]; [b_1, \dots, c_r]; z) = {}_{p-r}F_{q-r}([a_1, \dots, a_{p-r}]; [b_1, \dots, b_{q-r}]; z). $$

  The above rules applies even if any of the c_i happens to be zero or a negative integer (for details, see Luke in the list of references, p.42).

- $U(z) = {}_pF_q(a; b; z)$ satisfies a differential equation in z:

$$[\delta\,(\delta + b - 1) - z(\delta + a)]\, U(z) = 0, \quad \delta = z\,\frac{\mathrm{d}}{\mathrm{d}z},$$

  where $(\delta + a)$ and $(\delta + b)$ stand for $\prod_{i=1}^{p}(\delta + a_i)$ and $\prod_{j=1}^{q}(\delta + b_j)$, respectively. Thus, the order of this differential equation is $\max(p, q + 1)$ and the hypergeometric function is only one of its solutions. If $p < q + 1$, this differential equation has a regular singularity at $z = 0$ and an irregular singularity at $z = \infty$. If $p = q + 1$, the points $z = 0$, $z = 1$, and $z = \infty$ are regular singularities, thus explaining the convergence properties of the hypergeometric series.

  The analytic continuation for $p = q + 1, |z| \geq 1$, is defined by selecting the principal branch of this continuation (also denoted as ${}_pF_q(a; b; z)$) satisfying the condition $|\arg(1 - z)| < \pi$, the cut $[1, \infty)$ is drawn in the complex z-plane. In particular, the analytic continuation can be obtained by means of an integral representation (for details, see Prudnikov *et al.* in the references) or by the Meijer G function.

- References:

  - Y.L. Luke, "The Special Functions and Their Approximations", Vol. 1, Academic Press, New York, 1969.
  - A.P. Prudnikov, Yu.A. Brychkov and O.I. Marichev, "Integrals and Series", Vol. 3: More Special Functions, Gordon and Breach, 1990.
  - M. Abramowitz and I.A. Stegun, "Handbook of Mathematical Functions", Dover Publications, New York, 9th printing, 1970.

Changes:

- `hypergeom` is a new function.

`icontent` – the content of a polynomial with rational coefficients

`icontent(p)` computes the content of the polynomial `p` with integer or rational number coefficients, i.e., the gcd of its coefficients.

Call(s):

- `icontent(p)`

Parameters:

`p` — a polynomial or polynomial expression with integer or rational number coefficients

Return Value: a nonnegative integer or rational number, or `FAIL`

Related Functions: `coeff`, `content`, `factor`, `gcd`, `ifactor`, `igcd`, `ilcm`, `lcm`, `poly`, `polylib::primpart`

Details:

- `icontent(p)` calculates the content of a polynomial or polynomial expression with integer or rational coefficients, i.e., the greatest common divisor of the coefficients, such that `p/icontent(p)` has integral coefficients whose greatest common divisor is `1`. In particular, if `p` is itself an integer or a rational number, then `icontent` returns `abs(p)` (see example 1).
- If `p` is a polynomial or polynomial expression with integer coefficients, then the content is the greatest common divisor of the coefficients. If `p` is a polynomial or polynomial expression with rational coefficients, then the content is the greatest common divisor of the numerators of the coefficients divided by the least common multiple of the denominators (see example 2).
- If `p` is a polynomial expression, then it is first converted into a polynomial of domain type `DOM_POLY` using `poly`. If this conversion is not possible, then `icontent` returns `FAIL`.
- `icontent` returns an error message if not all coefficients of `p` are integers or rational numbers.
- `icontent` is a function of the system kernel.

Example 1. The first argument can be a polynomial or a polynomial expression. The following two calls of `icontent` are equivalent:

```
>> p := 6*x*y - 9*y^2 + 21:
   icontent(poly(p)), icontent(p)

                              3, 3
```

The result of `icontent` is always nonnegative:

```
>> icontent(2*x - 4), icontent(-2*x + 4)

                              2, 2
```

The content of a constant polynomial is its absolute value:

```
>> icontent(0), icontent(-2), icontent(poly(-2, [x]))

                             0, 2, 2
```

Example 2. The content of a polynomial with rational coefficients is a rational number in general:

```
>> q := 6/7*x*y - 9/4*y + 12:
   icontent(poly(q)), icontent(q)

                    3/28, 3/28
```

The polynomial divided by its content has integral coefficients whose greatest common divisor is 1:

```
>> q/icontent(q)

                 8 x y - 21 y + 112

>> icontent(%)

                         1
```

`if` – branch statement

`if-then-else-end_if` allows conditional branching in a program.

Call(s):

```
if condition1
 then casetrue1
<elif condition2 then casetrue2>
<elif condition3 then casetrue3>
<...>
<else casefalse>
end_if
```

```
_if(condition1, casetrue1, casefalse)
```

Parameters:

`condition1, condition2, ...`	—	Boolean expressions
`casetrue1, casetrue2, ..., casefalse`	—	arbitrary sequences of statements

Return Value: the result of the last command executed inside the `if` statement. `NIL` is returned if no command was executed.

Further Documentation: Chapter 17 of the MuPAD Tutorial.

Related Functions: `case`, `piecewise`

Details:

- If the Boolean expression `condition1` can be evaluated to `TRUE`, the branch `casetrue1` is executed and its result is returned. Otherwise, if `condition2` evaluates to `TRUE`, the branch `casetrue2` is executed and its result is returned etc. If all of the conditions evaluate to `FALSE`, the branch `casefalse` is executed and its result is returned.
- All conditions that are evaluated during the execution of the `if` statement must be reducible to either `TRUE` or `FALSE`. Conditions may be given by equations or inequalities, combined with the logical operators `and`, `or`, `not`. There is no need to enforce Boolean evaluation of equations and inequalities via `bool`. Implicitly, the `if` statement enforces "lazy" Boolean evaluation via the functions `_lazy_and` or `_lazy_or`, respectively. A condition leads to a runtime error if it cannot be evaluated to `TRUE` or `FALSE` by these functions. Cf. example 3.
- The keyword `end_if` may be replaced by the keyword `end`.
- The statement `if condition then casetrue else casefalse end_if` is equivalent to the function call `_if(condition, casetrue, casefalse)`.
- `_if` is a function of the system kernel.

Example 1. The `if` statement operates as demonstrated below:

```
>> if TRUE then YES else NO end_if,
   if FALSE then YES else NO end_if

                              YES, NO
```

The `else` branch is optional:

```
>> if FALSE then YES end_if

                                NIL
```

```
>> if FALSE
     then if TRUE
            then NO_YES
            else NO_NO
          end_if
     else if FALSE
            then YES_NO
            else YES_YES
          end_if
   end_if
```

```
YES_YES
```

Typically, the Boolean conditions are given by equations, inequalities or Boolean constants produced by system functions such as `isprime`:

```
>> for i from 100 to 600 do
     if 105 < i and i^2 <= 17000 and isprime(i) then
        print(expr2text(i)." is a prime")
     end_if;
     if i < 128 then
        if isprime(2^i - 1) then
           print("2^".expr2text(i)." - 1 is a prime")
        end_if
     end_if
   end_for:

                        "107 is a prime"

                     "2^107 - 1 is a prime"

                        "109 is a prime"

                        "113 is a prime"

                        "127 is a prime"

                     "2^127 - 1 is a prime"
```

Example 2. Instead of using nested `if-then-else` statements, the `elif` statement can make the source code more readable. However, internally the parser converts such statements into equivalent `if-then-else` statements:

```
>> hold(if FALSE then NO elif TRUE then YES_YES else YES_NO end_if)

                         if FALSE then
                           NO
                         else
                           if TRUE then
                             YES_YES
                           else
                             YES_NO
                           end_if
                         end_if
```

Example 3. If the condition cannot be evaluated to either TRUE or FALSE, then a runtime error is raised. In the following call, is(x > 0) produces UNKNOWN if no corresponding property was attached to x via assume:

```
>> if is(x > 0) then
     1
   else
     2
   end_if

 Error: Can't evaluate to boolean [if]
```

Note that Boolean conditions using <, <=, >, >= must not involve symbolic expressions:

```
>> if 1 < sqrt(2) then print("1 < sqrt(2)"); end_if

 Error: Can't evaluate to boolean [_less]

>> if 1 < float(sqrt(2)) then print("1 < float(sqrt(2))"); end_if

                      "1 < float(sqrt(2))"

>> if PI < 3.1416 then print("PI < 3.1416"); end_if

 Error: Can't evaluate to boolean [_less]
```

Example 4. This example demonstrates the correspondence between the functional and the imperative use of the if statement:

```
>> condition := 1 > 0: _if(condition, casetrue, casefalse)

                            casetrue

>> condition := 1 > 2: _if(condition, casetrue, casefalse)

                            casefalse

>> delete condition:
```

---

id – the identity map

id(x) evaluates and returns x.

Call(s):

- `id(x)`
- `id(x1, x2, ...)`

Parameters:

`x, x1, x2, ...` — arbitrary MuPAD objects

Return Value: the sequence of the input parameters.

Details:

- `id(x)` evaluates and returns `x`; `id(x1, x2, ...)` returns the evaluated arguments as an expression sequence; `id()` returns the void object `null()`.
- `id` is a function of the system kernel.

Example 1. `id` returns the evaluated arguments:

```
>> a := 2: id(a + 2)

                                    4

>> id(a, b, 4 + 2)

                                 2, b, 6
```

`id()` returns `null()`:

```
>> domtype(id())

                                DOM_NULL

>> delete a:
```

Example 2. `id` is useful when working with functional expressions:

```
>> f := 3*id + sin + 5*id^2 + exp@(-id^2): f(x)

                                    2           2
                 3 x + sin(x) + 5 x  + exp(- x )

>> D(f)

                                                2
                10 id + cos - 2 id exp@- id  + 3
```

```
>> delete f:
```

`ifactor` – factor an integer into primes

`ifactor(n)` computes the prime factorization $n = s \cdot p_1^{e_1} \cdots p_r^{e_r}$ of the integer n, where s is the sign of n, $p_1, \ldots, p_r$ are the distinct positive prime divisors of n, and $e_1, \ldots, e_r$ are positive integers.

Call(s):

- `ifactor(n <, UsePrimeTab>)`
- `ifactor(PrimeLimit)`

Parameters:

`n` — an arithmetical expression representing an integer

Options:

UsePrimeTab	—	look only for those prime factors that are stored in the internal prime table of the system
PrimeLimit	—	return the bound on the largest prime number in the prime table

Return Value: an object of domain type `Factored`, or a symbolic `ifactor` call.

Related Functions: `content`, `factor`, `Factored`, `icontent`, `igcd`, `ilcm`, `isprime`, `ithprime`, `nextprime`, `numlib::divisors`, `numlib::ecm`, `numlib::mpqs`, `numlib::pollard`, `numlib::prevprime`, `numlib::primedivisors`

Details:

- The result of `ifactor` is an object of domain type `Factored`. Let `f:=ifactor(n)` be such an object. Internally, it is represented by the list `[s, p1, e1, ..., pr, er]` of odd length $2r + 1$, where r is the number of distinct prime divisors of n. The p_i are not necessarily sorted by magnitude.

  You may extract the sign s, the primes p_i, as well as the exponents e_i by means of the index operator `[ ]`, so that `f[1] = s, f[2] = p1, f[3] = e1, ...`.

  For example, the command `f[2*i] $ i = 1..nops(f) div 2` returns all distinct prime divisors of n. The call `Factored::factors(f)` yields the same result, and `Factored::exponents(f)` returns a list of the exponents e_i for $1 \leq i \leq r$.

The factorization of 0, 1, and -1 yields the single factor 0, 1, and -1, respectively. In these cases, the internal representation is the list `[0]`, `[1]`, and `[-1]`, respectively.

The call `coerce(f,DOM_LIST)` returns the internal representation of a factored object, i.e., the list as described above.

Note that the result of `ifactor` is printed as an expression, and it is implicitly converted into an expression whenever it is processed further by other MuPAD functions. For example, the result of `ifactor(12)` is printed as `2^2*3`, which is an expression of type `"_mult"`.

See example 1 for illustrations, and the help page of `Factored` for more details.

- If you do not need the prime factorization of `n`, but only want to know whether it is composite or prime, use `isprime` instead, which is much faster.
- `ifactor` returns an error when the argument is a number but not an integer. A symbolic `ifactor` call is returned if the argument is not a number.

Option <`UsePrimeTab`>**:**

- Internally, MuPAD has stored a pre-computed table of all prime numbers up to a certain bound. `ifactor(n, `*`UsePrimeTab`*`)` looks only for prime factors that are stored in this internal prime number table, extracts them from `n`, and returns the undecomposed product of all other prime factors as a single factor. This is usually much faster than without the option *`UsePrimeTab`*, but it does not necessarily yield the complete prime factorization of `n`. See example 2.

Option <`PrimeLimit`>**:**

- `ifactor(`*`PrimeLimit`*`)` returns an integer, namely the bound on the largest prime number in the internal prime number table. The table contains all primes below this bound. The default values are: 1 000 000 on UNIX systems, and 300 000 on MacOS and Windows platforms.

  On UNIX platforms, the size of this table can be changed via the MuPAD command line flag `-L`.

Example 1. To get the prime factorization of 120, enter:

```
>> f := ifactor(120)

                               3
                              2  3 5
```

You can access the internal representation of this factorization using the index operator:

```
>> f[1];                          // the sign
   f[2*i]     $ i=1..nops(f) div 2; // the factors
   f[2*i + 1] $ i=1..nops(f) div 2; // the exponents

                                1

                             2, 3, 5

                             3, 1, 1
```

The internal representation of `f`, namely the list as described above, is returned by the following command:

```
>> coerce(f, DOM_LIST)

                      [1, 2, 3, 3, 1, 5, 1]
```

The result of `ifactor` is an object of domain type `Factored`:

```
>> domtype(f)

                            Factored
```

This domain implements some features for handling such objects. Some of them are described below.

You may extract the factors and exponents of the factorization also in the following way:

```
>> Factored::factors(f), Factored::exponents(f)

                      [2, 3, 5], [3, 1, 1]
```

You can ask for the type of the factorization:

```
>> Factored::getType(f)

                          "irreducible"
```

This output means that all p_i are prime. Other possible types are `"squarefree"` (see `polylib::sqrfree`) or `"unknown"`.

Multiplying factored objects preserves the factored form:

```
>> f2 := ifactor(12)

                            2
                           2  3

>> f*f2

                           5  2
                          2  3  5
```

It is important to note that you can apply nearly any function operating on arithmetical expressions to an object of domain type `Factored`. The result is usually not of this domain type:

```
>> expand(f);
   domtype(%)

                            120

                          DOM_INT
```

The function `type` implicitly converts an object of domain type `Factored` into an expression:

```
>> type(f)

                          "_mult"
```

For a detailed description of these objects, please refer to the help page of the domain `Factored`.

Example 2. The factorizations of `0`, `1`, and `-1` each have exactly one factor:

```
>> ifactor(0), ifactor(1), ifactor(-1)

                         0, 1, -1

>> map(%, coerce, DOM_LIST)

                       [0], [1], [-1]
```

The internal representation of the factorization of a prime number `p` is the list `[1, p, 1]`:

```
>> coerce(ifactor(5), DOM_LIST)

                         [1, 5, 1]
```

Example 3. The bound on the prime number table is:

```
>> ifactor(PrimeLimit)

                        1000000
```

We assign a large prime number to p:

```
>> p := nextprime(10^12)

                     1000000000039
```

Completely factoring the 36 digit number 6*p^3 takes some time; the second output line shows the time in seconds:

```
>> t := time():
   f := ifactor(6*p^3);
   (time() - t)/1000.0

                                  3
                 2 3 1000000000039

                         12.34

>> Factored::getType(f)

                     "irreducible"
```

Extracting only the prime factors in the prime table is much faster, but it does not yield the complete factorization; the factor p^3 remains undecomposed:

```
>> t := time():
   f := ifactor(6*p^3, UsePrimeTab);
   (time() - t)/1000.0

       2 3 1000000000117000000004563000000059319

                         0.21

>> Factored::getType(f)

                       "unknown"
```

Background:

- ifactor uses the elliptic curve method.

  ifactor is an interface to the kernel function stdlib::ifactor. It calls stdlib::ifactor with the given arguments and convert its result, which is the list [s, p1, e1, ..., pr, er] as described above, into an object of the domain type Factored.

You may directly call the kernel function stdlib::ifactor inside your routines, in order to avoid this conversion and to decrease the running time.

---

igamma – the incomplete Gamma function

igamma(a, x) represents the incomplete Gamma function $\int_x^\infty e^{-t}\, t^{a-1}\, \mathrm{d}t$.

Call(s):

- igamma(a, x)

Parameters:

a, x — arithmetical expressions

Return Value: an arithmetical expression.

Overloadable by: a, x

Side Effects: When called with a floating point argument, the function is sensitive to the environment variable DIGITS which determines the numerical working precision.

Related Functions: Ei, erfc, exp, fact, gamma, int

Details:

- A floating point value is returned if at least one of the arguments is a floating point value and both arguments are numerical satisfying the restrictions below. In all other cases, symbolic calls of igamma and/or other special functions may be returned.
- If a is real and positive, then floating point evaluation is possible for all positive real x.
- Further, if a is an integer multiple of 1/2, then floating point evaluation is possible for any complex x.
- Floating point evaluation may not be possible in other cases. In particular, if a is not a real number, then a symbolic call of igamma is returned.

- The following simplifications and rewriting rules are implemented:

$$\begin{array}{lcl}
\mathtt{igamma(a,0)} & \rightarrow & \mathtt{gamma(a)},\\
\mathtt{igamma(0,x)} & \rightarrow & \mathtt{Ei(x)},\\
\mathtt{igamma(1/2,x)} & \rightarrow & \mathtt{sqrt(PI)} * \mathtt{erfc(sqrt(x))},\\
\mathtt{igamma(1,x)} & \rightarrow & \mathtt{exp(-x)}.
\end{array}$$

  For real numerical values of a of `Type::Real` satisfying $|a| \leq 500$, the functional relation

$$\mathtt{igamma(a,x)} = \mathtt{x\hat{}(a-1)} * \mathtt{exp(-x)} + \mathtt{(a-1)} * \mathtt{igamma(a-1,x)}$$

  is used recursively to shift the first argument to the interval $0 \leq a \leq 1$. Thus rewriting in terms of `Ei`, `erfc`, and `exp` occurs if a is an integer multiple of $1/2$. Cf. example 1.

- The special value `igamma(a, infinity) = 0` for `a` $\neq$ `infinity` is implemented.

- The floating point evaluation is fast and numerically stable, if both arguments are real and positive. All other floating point evaluations may be subject to numerical cancellation! Cf. example 3.

Example 1. We demonstrate some calls with exact and symbolic input data:

```
>> igamma(2, 3), igamma(1/7, x), igamma(sqrt(2), 3)

                                             1/2
           4 exp(-3), igamma(1/7, x), igamma(2    , 3)
```

```
>> igamma(a, 4), igamma(1 + I, x^2 + 1), igamma(a, infinity)

                                        2
            igamma(a, 4), igamma(1 + I, x  + 1), 0
```

If the first argument a is a real numerical value with $|a| \leq 500$, the functional relations are used recursively until `igamma` is called with a first argument from the the interval $0 \leq a \leq 1$:

```
>> igamma(-1/10, 1), igamma(7/4, 1)

                                             3 igamma(3/4, 1)
   10 exp(-1) - 10 igamma(9/10, 1), exp(-1) + ----------------
                                                     4
```

If the first argument is an integer multiple of $1/2$, then complete rewriting in terms of `Ei`, `erfc`, and `exp` occurs:

```
>> igamma(-3, x), igamma(-5/2, x), igamma(8, x), igamma(13/2, 4)
```

```
        /  1      1      1  \   Ei(x)
exp(-x) | --- - ---- + ---- | - -----,
        | 6 x      2      3 |     6
        \       6 x    3 x  /

                                                1/2       1/2
           /    8         4         2   \   8 PI    erfc(x   )
   exp(-x) | ------- - ------- + ------ | - ------------------
           |     1/2       3/2      5/2 |           15
           \ 15 x      15 x      5 x    /

                             2        3        4       5
   , exp(-x) (5040 x + 2520 x  + 840 x  + 210 x  + 42 x  +

                                               1/2
      6    7          210979 exp(-4)   10395 PI    erfc(2)
   7 x  + x  + 5040), -------------- + -------------------
                            16                  64
```

Floating point values are computed for floating point arguments:

```
>> igamma(0.1, 4.0), igamma(7, 0.5), igamma(100, 100.0)

          0.004420083058, 719.9992783, 4.542198121e155
```

Example 2. Fast and numerically stable floating point evaluation is available for all real positive a and x:

```
>> igamma(1.0, 4.0), igamma(7.0, 10.2), igamma(12.3, 34.5)

             0.01831563889, 84.97892788, 361.781135
```

If a is not real, then no floating point evaluation is available:

```
>> igamma(1.0*I, 4.0), igamma(7.0 - 3.2*I, 10.2)

          igamma(1.0 I, 4.0), igamma(7.0 - 3.2 I, 10.2)
```

If a is not an integer multiple of $1/2$, then no floating point evaluation is available if x is not real and positive:

```
>> igamma(0.1, -4.0), igamma(3/4, 12.3 + 3.45*I)

          igamma(0.1, -4.0), igamma(3/4, 12.3 + 3.45 I)
```

If a is an integer multiple of $1/2$, then floating point evaluation is available for any complex x:

```
>> igamma(-3/2, -4.0), igamma(12, 12.3 + 3.45*I)

    2.363271801 - 2.925061501 I, 13266196.93 - 17206446.91 I
```

Example 3. The functional relation between igamma with different first arguments is used to "normalize" the returned expressions:

```
>> igamma(-8, x), igamma(7/3, x)

 Ei(x)           /     1          1           1          1
 ----- + exp(-x) | -------- - ------- - -------- + ------- -
 40320           |        2   40320 x          3          4
                 \ 40320 x              20160 x    6720 x

      1         1        1       1  \
   ------- + ------ - ----- + ---- |,
         5        6       7      8 |
   1680 x    336 x    56 x    8 x  /

                              /    1/3        \
   4 igamma(1/3, x)           | 4 x       4/3 |
   ---------------- + exp(-x) | ------ + x    |
          9                   \    3          /
```

Note that such expansions are also used in floating point evaluations if a and x are not real and positive. However, this representation may be numerically unstable if $|a|$ is large:

```
>> DIGITS := 10: igamma(-100, 100.0)

                             1.141161587e-221

>> DIGITS := 40: igamma(-100, 100.0)

         1.855592139673963122879821176407517691282e-246

>> delete DIGITS:
```

Changes:

- The normalization to first arguments in the range $0 \leq a \leq 1$ was restricted to a of `Type::Real` with $|a| \leq 500$.

---

igcd – the greatest common divisor of integers

`igcd(i1, i2, ...)` computes the greatest common divisor of the integers $i_1, i_2, \ldots$

Call(s):

- `igcd(i1, i2, ...)`

Parameters:

`i1, i2, ...` — arithmetical expressions representing integers

Return Value: a nonnegative integer, or a symbolic `igcd` call.

Related Functions: `content`, `div`, `divide`, `factor`, `gcd`, `gcdex`, `icontent`, `ifactor`, `igcdex`, `ilcm`, `lcm`, `mod`

Details:

- `igcd` computes the greatest common nonnegative divisor of a sequence of integers. `igcd` with a single numeric argument returns its absolute value. `igcd` returns `0` when all arguments are `0` or no argument is given.
- `igcd` returns an error message if one argument is a number but not an integer. If at least one of the arguments is `1` or `-1`, then `igcd` returns `1`. Otherwise, if one argument is not a number, then a symbolic `igcd` call is returned.
- `igcd` is a function of the system kernel.

Example 1. We compute the greatest common divisor of some integers:

```
>> igcd(-10, 6), igcd(6, 10, 15)

                              2, 1
```

```
>> a := 4420, 128, 8984, 488:
   igcd(a), igcd(a, 64)

                              4, 4
```

The next example shows some special cases:

```
>> igcd(), igcd(0), igcd(1), igcd(-1), igcd(2)

                          0, 0, 1, 1, 2
```

If one argument is not a number, then the result is a symbolic `igcd` call, except in some special cases:

```
>> delete x:
   igcd(a, x), igcd(1, x), igcd(-1, x)

              igcd(4420, 128, 8984, 488, x), 1, 1
```

```
>> type(igcd(a, x))
```

"igcd"

---

igcdex – the extended Euclidean algorithm for two integers

igcdex(x, y) computes the nonnegative greatest common divisor g of the integers x and y and integers s and t such that $g = sx + ty$.

Call(s):

- igcdex(x, y)

Parameters:

x, y — arithmetical expressions representing integers

Return Value: a sequence of three integers, or a symbolic igcdex call.

Related Functions: div, divide, factor, gcd, gcdex, ifactor, igcd, ilcm, lcm, mod, numlib::igcdmult

---

Details:

- igcdex(x, y) returns an expression sequence g, s, t with three elements, where g is the nonnegative greatest common divisor of x and y and s, t are integers such that $g = sx + ty$. These data are computed by the extended Euclidean algorithm for integers.

  igcdex(0, 0) returns the sequence 0, 1, 0. If x is nonzero, then igcdex(0, x) and igcdex(x, 0) return abs(x), 0, sign(x) and abs(x), sign(x), 0, respectively.

  If both x and y are nonzero integers, then the numbers s,t satisfy the inequalities $|s| < |y/g|$ and $|t| < |x/g|$.

- If one of the arguments is a number but not an integer, then igcdex returns an error message. If some argument is not a number, then igcdex returns a symbolic igcdex call.

- The function numlib::igcdmult is an extension of igcdex for more than two arguments.

- igcdex is a function of the system kernel.

Example 1. We compute the greatest common divisor of some integers:

```
>> igcdex(-10, 6)
```

```
2, 1, 2
```

```
>> igcdex(3839882200, 654365735423132432848652680)
```

```
109710920, -681651885490791809, 4
```

The returned numbers satisfy the described equation:

```
>> [g, s, t] := [igcdex(9, 15)];
   g = s*9 + t*15
```

```
[3, 2, -1]
```

```
3 = 3
```

If one argument is not a number, the result is the a symbolic `igcdex` call:

```
>> delete x:
   igcdex(4, x)
```

```
igcdex(4, x)
```

`ilcm` – the least common multiple of integers

`ilcm(i1, i2, ...)` computes the least common multiple of the integers $i_1, i_2, \ldots$

Call(s):

- `ilcm(i1, i2, ...)`

Parameters:

`i1, i2, ...` — arithmetical expressions representing integers

Return Value: a nonnegative integer, or a symbolic `ilcm` call.

Related Functions: `content`, `factor`, `gcd`, `gcdex`, `icontent`, `ifactor`, `igcd`, `igcdex`, `lcm`

Details:

- `ilcm` computes the least common nonnegative multiple of a sequence of integers. `ilcm` with a single numeric argument returns its absolute value. `ilcm` returns `1` when all arguments are `1` or `-1` or no argument is given.
- `ilcm` returns an error message when one of the arguments is a number but not an integer. If at least one of the arguments is `0`, then `ilcm` returns `0`. Otherwise, if one argument is not a number, then a symbolic `ilcm` call is returned.
- `ilcm` is a function of the system kernel.

Example 1. We compute the least common multiple of some integers:

```
>> ilcm(-10, 6), ilcm(6, 10, 15)
```

```
30, 30
```

```
>> a := 4420, 128, 8984, 488:
   ilcm(a), ilcm(a, 64)
```

```
9689064320, 9689064320
```

The next example shows some special cases:

```
>> ilcm(), ilcm(0), ilcm(1), ilcm(-1), ilcm(2)
```

```
1, 0, 1, 1, 2
```

If one argument is not a number, then the result is a symbolic `ilcm` call, except in some special cases:

```
>> delete x:
   ilcm(a, x), ilcm(0, x)
```

```
ilcm(4420, 128, 8984, 488, x), 0
```

```
>> type(ilcm(a, x))
```

```
"ilcm"
```

`in` – membership

`x in set` is the "element of" relation. Further, the keyword `in` may also be used in combination with the keywords `for` and `$`, where it means "iterate over all operands."

Call(s):

- `x in set`
- `_in(x, set)`
- `for y in object do ... end_for`
- `f(y) $ y in object`

Parameters:

`x`	—	an arbitrary MuPAD object
`set`	—	a set or an object of set-like type
`y`	—	an identifier or a local variable (`DOM_VAR`) of a procedure
`object, f(y)`	—	arbitrary MuPAD objects

Overloadable by: `x, set`

Return Value: `x in set` just returns the input.

Related Functions: `_seqin`, `bool`, `contains`, `for`, `has`, `is`

Details:

- `x in set` is the MuPAD notation for the statement "x is a member of `set`."
- In conjunction with one of the keywords `for` or `$`, the meaning changes to "iterate over all operands of the object". See `for` and `$` for details. Cf. example 6.
- Apart from the usage with `for` and `$`, the statement `x in object` is equivalent to the function call `_in(x, object)`.
- `x in set` is just evaluated to itself. `expand(x in set)` tries to return an equivalent expression without using the operator `in`, as described in the following paragraphs.
- For sets of type `DOM_SET`, set unions, differences and intersections, `x in set` is expanded to an equivalent Boolean expression of equations and expressions involving `in`. Cf. example 1.
- If `set` is a solution set of a single equation in one unknown, given by a symbolic call to `solve`, expanding `in` returns a Boolean condition that is equivalent to `x` being a solution. Cf. example 2.
- If `set` is a `RootOf` expression, expanding `in` returns a Boolean condition that is equivalent to `x` being a root of the corresponding equation. Cf. example 3.

- The function `bool` and every function that uses boolean evaluation can also handle many logical expressions involving `in`. Cf. example 4.

- The function `is` handles various logical statements involving `in`, including a variety of types for the parameter `set` which are not handled by `in` itself. Cf. example 5 for a few typical cases.

- Apart from the usual overloading mechanism by the first argument of an `in` call, `in` can be overloaded by its second argument, too. This argument must define the slot `"set2expr"` for this purpose. The slot will be called with the arguments `set, x`.

Example 1. `x in {1, 2, 3}` is expanded into an equivalent statement involving `=` and `or`:

```
>> expand(x in {1, 2, 3})

                    x = 1 or x = 2 or x = 3
```

The same happens if you replace `x` by a number, because Boolean expressions are only evaluated inside certain functions such as `bool` or `is`:

```
>> expand(1 in {1, 2, 3}), bool(1 in {1, 2, 3}), is(1 in {1, 2, 3})

              1 = 1 or 1 = 2 or 1 = 3, TRUE, TRUE
```

If only some part of the expression can be simplified this way, the returned expression can contain unevaluated calls to `in`:

```
>> expand(x in {1, 2, 3} union A)

                x in A or x = 1 or x = 2 or x = 3
```

Example 2. For symbolic calls to `solve` representing the solution set of a single equation in one unknown, `in` can be used to check whether a particular value lies in the solution set:

```
>> solve(x^2 = 2^x, x); expand(2 in %), bool(2 in %)

                             2    x
                      solve(x  - 2  = 0, x)

                           0 = 0, TRUE
```

Example 3. `in` can be used to check whether a value is a member of the solution set represented by a `RootOf` expression:

```
>> r := RootOf(x^2 - 1, x);
   expand(1 in r), bool(1 in r), expand(2 in r), bool(2 in r)

                                  2
                         RootOf(x  - 1, x)

                     0 = 0, TRUE, 3 = 0, FALSE

>> expand((y - 1) in RootOf(x^2 - 1 - y^2 + 2*y, x))

                           2          2
                    2 y - y  + (y - 1)  - 1 = 0

>> expand(%)

                               0 = 0

>> delete r:
```

Example 4. Expressions with operator `in` are boolean expressions: they can be used like equations or inequalities.

```
>> if 2 in {2, 3, 5} then "ok" end

                               "ok"
```

Example 5. The MuPAD function `is` can investigate membership of objects in infinite sets. It respects properties of identifiers:

```
>> is(123 in Q_), is(2/3 in Q_)

                            TRUE, TRUE

>> assume(p, Type::Prime): is(p in Z_), is(p in Type::NonNegative)

                            TRUE, TRUE

>> unassume(p):
```

Example 6. In conjunction with `for` and `$`, `y in object` iterates y over all operands of the object:

```
>> for y in [1, 2] do
     print(y)
   end_for:

                                 1

                                 2

>> y^2 + 1 $ y in a + b*c + d^2

                      2       2  2       4
                     a  + 1, b  c  + 1, d  + 1

>> delete y:
```

Changes:

- `_in` now just returns the unevaluated function call; a new slot `"expand"` has been introduced to provide the old functionality.

`indets` – the indeterminates of an expression

`indets(object)` returns the indeterminates contained in `object`.

Call(s):

- `indets(object)`
- `indets(object, `*PolyExpr*`)`
- `indets(object, `*RatExpr*`)`

Parameters:

`object` — an arbitrary object

Options:

PolyExpr — return a set of arithmetical expressions such that `object` is a polynomial expression in the returned expressions

RatExpr — return a set of arithmetical expressions such that `object` is a rational expression in the returned expressions

Return Value: a set of arithmetical expressions.

Overloadable by: `object`

Related Functions: `collect`, `domtype`, `op`, `poly`, `rationalize`, `type`, `Type::Indeterminate`, `Type::PolyExpr`, `Type::RatExpr`

Details:

- `indets(object)` returns the indeterminates of `object` as a set, i.e., the identifiers without a value that occur in `object`, with the exception of those identifiers occurring in the `0`th operand of a subexpression of `object` (see example 1).
- `indets` regards the special identifiers `PI`, `EULER`, `CATALAN` as indeterminates, although they represent constant real numbers. If you want to exclude these special identifiers, use `indets(object) minus Type::ConstantIdents` (see example 1).
- If `object` is a polynomial, a function environment, a procedure, or a built-in kernel function, then `indets` returns the empty set. See example 2.
- `indets` is a function of the system kernel.

Option <`PolyExpr`>**:**

- With this option, `object` is considered as a polynomial expression. Non-polynomial subexpressions, such as `sin(x)`, `x^(1/3)`, `1/(x+1)`, or `f(a, b)`, are considered as indeterminates and are included in the returned set. However, subexpressions such as `f(2, 3)` are considered as constants even when the identifier `f` has no value. The philosophy behind this is that the expression is constant because the operands are constant (see example 1).
- If `object` is an array, a list, a set, or a table, then `indets` returns a set of arithmetical expressions such that each entry of `object` is a polynomial expression in these expressions. See example 2.

Option <`RatExpr`>**:**

- With this option, `object` is considered as a rational expression. Similar to *PolyExpr*, non-rational subexpressions are considered as indeterminates (see example 1).

Example 1. Consider the following expression:

```
>> delete g, h, u, v, x, y, z:
   e := 1/(x[u] + g^h) - f(1/3) + (sin(y) + 1)^2*PI^3 + z^(-3) * v^(1/2)

                                                        1/2
            1         3             2              v
        --------- + PI  (sin(y) + 1)  - f(1/3) + ----
         h                                          3
        g  + x[u]                                  z
```

```
>> indets(e)

                   {g, h, u, v, x, y, z, PI}
```

Note that the returned set contains `x` and `u` and not, as one might expect, `x[u]`, since internally `x[u]` is converted into the functional form `_index(x, u)`. Moreover, the identifier `f` is not considered an indeterminate, since it is the `0`-th operand of the subexpression `f(1/3)`.

Although `PI` mathematically represents a constant, it is considered an indeterminate by `indets`. Use `Type::ConstantIdents` to circumvent this:

```
>> indets(e) minus Type::ConstantIdents

                     {g, h, u, v, x, y, z}
```

The result of `indets` is substantially different if one of the two options is specified:

```
>> indets(e, RatExpr)

                                   h          1/2
                 {z, PI, sin(y), g  , x[u], v    }
```

Indeed, `e` is a rational expression in the "indeterminates" `z, PI, sin(y), g^h, x[u], v^(1/2)`: `e` is built from these atoms and the constant expression `f(1/3)` by using only the rational operations + , -, *, /, and ^ with integer exponents. Similarly, `e` is built from `PI,sin(y),z^(-3),1/(g^h+x[u]),v^(1/2)` and the constant expression `f(1/3)` using only the polynomial operations +, -, *, and ^ with nonnegative integer exponents:

```
>> indets(e, PolyExpr)

                {               1       1       1/2 }
                { PI, sin(y), --, ---------, v      }
                {               3   h               }
                {              z   g  + x[u]        }
```

Example 2. `indets` also works for various other data types. Polynomials and functions are considered to have no indeterminates:

```
>> delete x, y:
   indets(poly(x*y, [x, y])), indets(sin), indets(x -> x^2+1)

                            {}, {}, {}
```

For container objects, `indets` returns the union of the indeterminates of all entries:

```
>> indets([x, exp(y)]), indets([x, exp(y)], PolyExpr)

                       {x, y}, {x, exp(y)}
```

For tables, only the indeterminates of the entries are returned; indeterminates in the indices are ignored:

```
>> indets(table(x = 1 + sin(y), 2 = PI))

                             {y, PI}
```

Background:

- If `object` is an element of a library domain `T` that has a slot `"indets"`, then the slot routine `T::indets` is called with `object` as argument. This can be used to extend the functionality of `indets` to user-defined domains. If no such slot exists, then `indets` returns the empty set.

`indexval` – indexed access to arrays and tables without evaluation

`indexval(x, i)` and `indexval(x, i1, i2, ...)` yields the entry of `x` corresponding to the indices `i` and `i1, i2, ...`, respectively, without evaluation.

Call(s):

- `indexval(x, i)`
- `indexval(x, i1, i2, ...)`

Parameters:

`x`	—	essentially a table or an array, however, also allowed: a list, a finite set, an expression sequence, or a character string
`i, i1, i2, ...`	—	indices. For most "containers" `x`, indices must be integers. If `x` is a table, arbitrary MuPAD objects can be used as indices.

Return Value: the entry of `x` corresponding to the index. When `x` is a table or an array, the returned entry is not evaluated again.

Overloadable by: `x`

Related Functions: `:=`, `_assign`, `_index`, `array`, `contains`, `DOM_ARRAY`, `DOM_LIST`, `DOM_SET`, `DOM_STRING`, `DOM_TABLE`, `op`, `table`

Details:

- The three calls `indexval(x, i)`, `_index(x, i)`, and `x[i]` all return the element of index `i` in the array or table `x`. In contrast to `_index` and the equivalent index operator `[]`, however, `indexval` returns the corresponding entry without evaluating it. This is sometimes desirable for efficiency reasons.
- The arguments `i` or `i1, i2, ...` must be a valid indices of `x`, otherwise an error message is printed (see example 3). When several indices `i1, i2, ...` are given, they are interpreted as a higher-dimensional index (see example 4).
- The first argument `x` may also be a list, a set, a string, or an expression sequence. However, in these cases `indexval` behaves exactly like `_index` and the index operator `[]`: it returns the evaluation of the corresponding element. In particular, `indexval` does not flatten its first argument.
- For all other basic domains, `indexval` behaves exactly like `_index`: either an error occurs, or a symbolic `indexval` call is returned (see example 3).
- `indexval` does not work with matrices in the current version. However, the function `_index` return an entry of a matrix unevaluated. NOTE
- `indexval` is a function of the system kernel.

Example 1. `indexval` works with tables:

```
>> T := table("1" = a, Be = b, ‘+‘ = a + b):
   a := 1: b := 2:
   indexval(T, Be), indexval(T, "1"), indexval(T, ‘+‘)

                          b, a, a + b
```

In contrast `_index` evaluates returned entries:

```
>> _index(T, Be), _index(T, "1"), _index(T, ‘+‘)

                            2, 1, 3
```

The next input line has the same meaning as the last:

```
>> T[Be], T["1"], T[`+`]
```

2, 1, 3

`indexval` works with arrays, too. The behavior is the same, but the indices must be positive integers:

```
>> delete a, b:
   A := array(1..2, 1..2, [[a, a + b], [a - b, b]]):
   a := 1: b := 2:
   indexval(A, 2, 2), indexval(A, 1, 1), indexval(A, 1, 2)
```

b, a, a + b

```
>> _index(A, 2, 2), _index(A, 1, 1), _index(A, 1, 2)
```

2, 1, 3

```
>> A[2, 2], A[1, 1], A[1, 2]
```

2, 1, 3

```
>> delete A, T, a, b:
```

Example 2. However, there is no difference between `indexval` and `_index` for all other valid objects, e.g., lists:

```
>> delete a, b:
   L := [a, b, 2]:
   b := 5:
   L[2], _index(L, 2), indexval(L, 2), op(L, 2)
```

5, 5, 5, 5

Similarly, there is no difference when the first argument is an expression sequence (which is not flattened by `indexval`):

```
>> delete a, b: S := a, b, 2:
   b := 5:
   S[2], _index(S, 2), indexval(S, 2), op(S, 2)
```

5, 5, 5, 5

```
>> delete L, S, a, b:
```

Example 3. If the second argument is not a valid index, an error occurs:

```
>> A := array(1..2, 1..2, [[a, b], [a, b]]):
   indexval(A, 3)

 Error: Index dimension mismatch [array]

>> indexval(A, 1, 0)

 Error: Illegal argument [array]

>> indexval("12345", 5)

 Error: Invalid index [string]
```

However, the result of `indexval` can also be a symbolic `indexval` call:

```
>> T := table(1 = a, 2 = b):
   indexval(T, 3)

                          indexval(T, 3)

>> delete X, i:
   indexval(X, i)

                          indexval(X, i)

>> delete A, T:
```

Example 4. For arrays the number of indices must be equal to the number of dimensions of the array:

```
>> A := array(1..2, 1..2, [[a, b], [a, b]]):
   a := 1: b := 2:
   indexval(A, 1, 2), indexval(A, 2, 1)

                               b, a
```

Otherwise an error occurs:

```
>> indexval(A, 1)

 Error: Index dimension mismatch [array]
```

Tables can have expression sequences as indices, too:

```
>> delete a, b:
   T := table((1, 1) = a, (2, 2) = b):
   a := 1: b := 2:
   indexval(T, 1, 1), indexval(T, 2, 2)
```

```
a, b
```

```
>> delete A, T, a, b:
```

`interpolate` – polynomial interpolation

`interpolate` computes an interpolating polynomial through data over a rectangular grid.

Call(s):

- `interpolate(xList, yList, X <, F>)`
- `interpolate(nodes, values, ind <, F>)`

Parameters:

`xList` — the nodes: a list `[x1, x2, ...]` of distinct arithmetical expressions

`yList` — the values: a list `[y1, y2, ...]` of arithmetical expressions. This list must have the same length as `xList`.

`X` — an indeterminate or an arithmetical expression. An indeterminate is either an identifier (of domain type `DOM_IDENT`) or an indexed identifier (of type `"_index"`).

`nodes` — a list $[L_1, \ldots, L_d]$ of d lists L_i defining a d-dimensional rectangular grid

$$\{(x_1, \ \ldots \ , x_d) \ ; \ x_1 \in L_1 \ , \ \ldots \ , \ x_d \in L_d\} \ .$$

The lists L_i may have different lengths $n_i = \texttt{nops}(L_i)$. The elements of each L_i must be distinct.

`values` — a d-dimensional $\texttt{array}(1..n_1, \ldots, 1..n_d, [\ldots])$ associating a value with each grid point:

$$[L_1[i_1], \ldots, L_d[i_d]] \quad \longrightarrow \quad \texttt{values}[i_1, \ldots, i_d] \ ,$$
$$i_1 = 1, .., n_1 \ , \ \ldots \ , \ i_d = 1, .., n_d \ .$$

`ind` — a list of d indeterminates or arithmetical expressions. Indeterminates are either identifiers (of domain type `DOM_IDENT`) or indexed identifiers (of type `"_index"`).

Options:

`F` — either ***Expr*** or any field of category `Cat::Field`

Return Value: An interpolating polynomial P of domain type `DOM_POLY` in the indeterminates specified by `ind` over the coefficient field `F` is returned. The elements in `ind` that are not indeterminates but arithmetical expressions are not used as indeterminates in P, but enter its coefficients: the polynomial is "evaluated" at these points. If no element of `ind` is an indeterminate, the value of the polynomial at the point specified by `ind` is returned. This is an element of the field `F` or an arithmetical expression if `F` = ***Expr***.

Related Functions: `genpoly`, `numeric::cubicSpline`, `numeric::cubicSpline2d`, `poly`

Details:

- The call `interpolate(xList, yList, X <, F>)` with `xList` = `[x1, ..., x.n]` and `yList` = `[y1, ..., y.n]` returns the polynomial of degree less than n in the variable `X` which interpolates the points `(x1, y1)`, ... , `(x.n, y.n)`.

  This call with a 1-dimensional grid `xList` is equivalent to the corresponding 'multi-dimensional' call `interpolate([xList], array(1..n, [yList]), [X] <, F>)`.

- For d-dimensional interpolation, assume that indeterminates `ind` $= [X_1, \ldots, X_d]$ are specified. The interpolating polynomial $P = \texttt{poly}(\ldots, [X_1, \ldots, X_d], F)$ satisfies

  $$\texttt{evalp}(P, X_1 = L_1[i_1], \ldots, X_d = L_d[i_d]) = \texttt{value}[i_1, \ldots, i_d]$$

  for all points $[L_1[i_1], \ldots, L_d[i_d]]$ in the grid. P is the polynomial of minimal degree satisfying the interpolation conditions, i.e., $\texttt{degree}(P, X_i) < n_i$.

- If only interpolating values at concrete numerical points $X_1 = v_1, \ldots, X_d = v_d$ are required, we recommend not to compute P with symbolic indeterminates `ind` $= [X_1, \ldots, X_d]$ and then evaluate $P(v_1, \ldots, v_d)$. It is faster to compute this value directly by `interpolate` with `ind` $= [v_1, \ldots, v_d]$. Cf. examples 1 and 3.

Option <F>:

- The returned polynomial is of type `poly(..., F)`.
- For the default field ***Expr***, all input data may be arbitrary MuPAD expressions. Standard arithmetic over such expressions is used to compute the polynomial.
- For `F` $\neq$ ***Expr***, the grid nodes as well as the entries of `values` must be elements of `F` or must be convertible to such elements. Conversion of the input data to elements of `F` is done automatically.

Example 1. We consider a 1-dimensional interpolation problem. To each node x_i, a value y_i is associated. The interpolation polynomial P with $P(x_i) = y_i$ is:

```
>> xList := [1, 2, 3]:
   yList := [y1, y2, y3]:
   P := interpolate(xList, yList, X)

     / / y1         y3 \  2   /   5 y1           3 y3 \
 poly| | -- - y2 + -- | X  + | - ---- + 4 y2 - ---- | X +
     \ \  2          2 /      \     2              2  /

                                \
    (3 y1 - 3 y2 + y3), [X] |
                                /
```

The evaluation of P at the point $X = 5/2$ is given by:

```
>> evalp(P, X = 5/2)

                        3 y2   y1   3 y3
                        ---- - -- + ----
                          4     8     8
```

This value can also be computed directly without the symbolic polynomial:

```
>> interpolate(xList,  yList, 5/2)

                        3 y2   y1   3 y3
                        ---- - -- + ----
                          4     8     8

>> delete xList, yList, P:
```

Example 2. We demonstrate multi-dimensional interpolation. Consider data over the following 2-dimensional 2×3 grid:

```
>> XList := [1, 2]: YList := [1, 2, 3]:
   values := array(1..2, 1..3, [[1, 2, 3], [3, 2, 1]]):
   P := interpolate([XList, YList], values, [X, Y])

              poly(- 2 X Y + 4 X + 3 Y - 4, [X, Y])
```

Next, interpolation over a 3-dimensional $2 \times 3 \times 2$ grid is demonstrated:

```
>> L1 := [1, 2]: L2 := [1, 2, 3]: L3 := [1, 2]:
   values := array(1..2, 1..3, 1..2,
      [[[1, 4], [1, 2], [3, 3]], [[1, 4], [1, 3], [4, 0]]]):
   interpolate([L1, L2, L3], values, [X, Y, Z])
```

```
              2              2
 poly(- 3 X Y  Z + 7/2 X Y  + 10 X Y Z - 23/2 X Y - 7 X Z +

                2         2
    8 X + 7/2 Y  Z - 3 Y  - 27/2 Y Z + 12 Y + 13 Z - 11,

    [X, Y, Z])
```

```
>> delete XList, values, P, L1, L2, L3:
```

Example 3. We interpolate data over a 2-dimensional grid:

```
>> n1 := 4: L1 := [i $ i = 1..n1]:
   n2 := 5: L2 := [i $ i = 1..n2]:
   f := (X, Y) -> 1/(1 + X^2 + Y^2):
   values := array(1..n1, 1..n2,
                   [[f(L1[i], L2[j]) $ j=1..n2] $ i=1..n1]):
```

First, we compute the symbolic polynomial:

```
>> P := interpolate([L1, L2], values, [X, Y])

                          3   4                   3   3
 poly(- 5563/23108085 X  Y  + 16376/4621617 X  Y  -

    ... -

    4401895/3081078 Y + 4199983/2567565, [X, Y])
```

Fixing the value $Y = 2.5$, this yields a polynomial in X.

```
>> evalp(P, Y = 2.5)

                          3                    2
 poly(0.0007372500794 X  - 0.002155538175 X  -

    0.03076935248 X + 0.1533997618, [X])
```

It can also be computed directly by using an evaluation point for the indeterminate Y:

```
>> interpolate([L1, L2], values, [X, 2.5])

                          3                    2
 poly(0.0007372500794 X  - 0.002155538175 X  -

    0.03076935248 X + 0.1533997618, [X])
```

If all indeterminates are replaced by evaluation points, the corresponding interpolation value is returned:

```
>> interpolate([L1, L2], values, [1.2, 2.5])

                    0.114646532

>> delete n1, n2, f, values, P:
```

Example 4. We demonstrate interpolation over a special coefficient field. Consider the following data over a 2-dimensional 2×3 grid:

```
>> XList := [3, 4]: YList := [1, 2, 3]:
   values := array(1..2, 1..3, [[0, 1, 2], [3, 2, 1]]):
```

With the following call, these data are converted to integers modulo 7. Arithmetic over this field is used:

```
>> F := Dom::IntegerMod(7):
   P := interpolate([XList, YList], values, [X, Y], F)

       poly(5 X Y + 5 X + 5, [X, Y], Dom::IntegerMod(7))
```

Evaluation of P at grid points reproduces the associated values converted to the field:

```
>> evalp(P, X = XList[2], Y = YList[3]) = F(values[2, 3])

                    1 mod 7 = 1 mod 7

>> delete XList, YList, values, F, P:
```

Background:

- For a d-dimensional rectangular grid

$$\{(x_1, \ldots, x_d)\ ;\ x_1 \in L_1\ ,\ \ldots\ ,\ x_d \in L_d\}$$

specified by the lists

$$L_j = [x_{j1}, \ldots, x_{jn_j}]\ , \quad j = 1, \ldots, d$$

with associated values

$$P(x_{1i_1}, \ldots, x_{di_d}) = v_{i_1,\ldots,i_d},$$

the interpolating polynomial in the indeterminates $X_1, \ldots, X_d$ is given by

$$P(X_1, \ldots, X_d) = \sum_{i_1=1}^{n_1} \cdots \sum_{i_d=1}^{n_d} v_{i_1,\ldots,i_d} \times p_{1i_1}(X_1) \times \cdots \times p_{di_d}(X_d)$$

with the Lagrange polynomials

$$p_{jk}(X) = \prod_{\substack{l=1,\ldots,n_j\\ l\neq k}} \frac{X - x_{jl}}{x_{jk} - x_{jl}}\ , \quad j = 1,\ldots,d, \quad k = 1,\ldots,n_j$$

associated with the k-th node of the j-th coordinate.

Changes:

- `interpolate` used to be `numeric::lagrange`.
- The simplified input syntax `interpolate(xList, yList, x <, F>)` was introduced for 1-dimensional grids.

`infinity` – **infinity**

`infinity` represents the infinite point on the positive real semi-axis.

Related Functions: `complexInfinity`, `undefined`

Details:

- `infinity` is an element of the domain `stdlib::Infinity`. It may be used in arithmetical operations. Some system functions accept `infinity` as a parameter or return it as a result.

Example 1. `infinity` can be used in arithmetical operations with real numbers:

```
>> 7*infinity + 3, -3.0*infinity, 1/infinity,
   infinity*infinity, infinity^2, sqrt(infinity)

      infinity, -infinity, 0, infinity, infinity, infinity
```

Arithmetic with complex numbers or symbolic objects yields symbolic expressions:

```
>> I*infinity + b

                        b + I infinity
```

The arithmetic responds to properties:

```
>> assume(a > 0): a*infinity

                           infinity
```

```
>> assume(a < 0): a*infinity

                            -infinity

>> unassume(a): a*infinity

                            a infinity
```

Cancellation of infinities yields `undefined`:

```
>> infinity - infinity, infinity/infinity

                       undefined, undefined
```

Some system functions accept `infinity` as a parameter or return it as result:

```
>> exp(infinity), sum(1/n, n = 1..infinity),
   int(exp(-x^2), x = -infinity..infinity),
   limit(x, x = infinity)

                                       1/2
              infinity, infinity, PI   , infinity
```

`info` – prints short information

`info(object)` prints short information about `object`.

`info()` prints a list of all available MuPAD libraries.

Call(s):

- `info(object)`
- `info()`

Parameters:

`object` — any MuPAD object

Return Value: the void object `null()` of type `DOM_NULL`.

Side Effects: The formatting of the output of `info` is sensitive to the environment variable `TEXTWIDTH`.

Related Functions: `help`, `export`, `print`, `setuserinfo`, `userinfo`

Details:

- `info` prints a short descriptive information about `object`.
- If `object` is a domain, additional information is given about the methods of the domain.
- A call to `info` without arguments prints the names of all available system libraries.
- Users can add information about their own functions and domains by `overloading info`. If `object` is a user-defined domain or function environment providing a slot `"info"`, whose value is a string, then the call `info(object)` prints this string. See example 2.

Example 1. With `info()`, you obtain a list of all libraries:

```
>> info()

 -- Libraries:
 Ax,         Cat,       Dom,       Network,   RGB,
 Series,     Type,      adt,       combinat,  detools,
 fp,         generate,  groebner,  import,    intlib,
 linalg,     linopt,    listlib,   matchlib,  module,
 numeric,    numlib,    ode,       orthpoly,  output,
 plot,       polylib,   prog,      property,  solvelib,
 specfunc,   stats,     stdlib,    stringlib, student,
 transform
```

The next example shows information about the library `property`:

```
>> info(property)

 Library 'property': properties of identifiers

 -- Interface:
 property::Null,   property::hasprop, property::implies,
 property::simpex

 -- Exported:
 assume, getprop, is, unassume
```

`info` prints information about preferences:

```
>> info(Pref::promptString)

 A character string to be displayed as a prompt.
```

If no more information is available, a short type description is given:

```
>> info(a + b):
   info([a, b]):

 a + b -- an expression of type "_plus"
 [a, b] -- of domain type 'DOM_LIST'
```

Example 2. `info` prints information about a function environment:

```
>> info(sqrt)

 sqrt -- the square root
```

`sqrt` is a function environment and has a slot named `"info"`:

```
>> domtype(sqrt), sqrt::info

             DOM_FUNC_ENV, "sqrt -- the square root"
```

User-defined procedures can contain short information. By default, `info` does only return some general information:

```
>> f := x -> x^2:  info(f):

 f -- a procedure of domain type 'DOM_PROC'
```

To improve this, we embed the function `f` into a function environment and store an information string in its `"info"` slot:

```
>> f := funcenv(f):
   f::info := "f -- the squaring function":
   info(f)

 f -- the squaring function

>> delete f:
```

Background:

- If the argument `object` of `info` is a domain, then the call `info(object)` first prints the entry `"info"`, which must be a string. Then the entry `"interface"`, which must be a set of identifiers, is used to display all public methods, and the entry `"exported"`, which is a set of identifiers created by `export`, is used to display all exported methods.

Changes:

- `info` now prints more information like e.g. the type of an expression.

`input` – interactive input of MuPAD objects

`input` allows interactive input of MuPAD objects.

Call(s):

- `input(<prompt1>)`
- `input(<prompt1,> x1, <prompt2,> x2, ...)`

Parameters:

`prompt1, prompt2, ...`	—	input prompts: character strings
`x1, x2, ...`	—	identifiers

Return Value: the last input

Related Functions: `finput`, `fprint`, `fread`, `ftextinput`, `print`, `read`, `text2expr`, `textinput`, `write`

Details:

- `input()` displays the prompt "`Please enter expression :`" and waits for input by the user. The input, terminated by pressing the <Return> key, is parsed and returned *unevaluatedly.*
- `input(prompt1)` uses the character string `prompt1` instead of the default prompt "`Please enter expression :`".
- `input(<prompt1,> x1)` assigns the input to the identifier `x1`. The default prompt is used, if no prompt string is specified.
- Several objects can be read with a single `input` command. Each identifier in the sequence of arguments makes `input` return a prompt, waiting for input to be assigned to the identifier. A character string preceding the identifier in the argument sequence replaces the default prompt (see example 2). Arguments that are neither prompt strings nor identifiers are ignored.
- The identifiers `x1` etc. may have values. These are overwritten by `input`.
- `input` only parses the input objects for syntactical correctness. It does not evaluate them. Use `eval` to evaluate the results (see example 3).
- `input` is a function of the system kernel.

Example 1. The default prompt is displayed. The input is returned without evaluation:

```
>> input()

 Please enter expression : << 1 + 2 >>

                              1 + 2
```

A character string is used as a prompt:

```
>> input("enter a number: ")

 enter a number: << 5 >>

                                5
```

The input may be assigned to an identifier:

```
>> input(x)

 Please enter expression : << 5 >>

                                5

>> x

                                5
```

A user-defined prompt is used, the input is assigned to an identifier:

```
>> input("enter a number: ", x)

 enter a number: << 6 >>

                                6

>> x

                                6

>> delete x:
```

Example 2. If several objects are to be read, for each object a separate prompt can be defined:

```
>> input("enter a matrix: ", A, "enter a vector: ", x)
```

```
 enter a matrix: << matrix([[a11, a12], [a21, a22]]) >>
 enter a vector: << matrix([x1, x2]) >>

                       matrix([x1, x2])

>> A, x

                     +-           -+  +-     -+
                     |  a11, a12  |   |  x1   |
                     |            |,  |       |
                     |  a21, a22  |   |  x2   |
                     +-           -+  +-     -+

>> delete A, x:
```

Example 3. The following procedure asks for an expression and a variable. After interactive input, the derivative of the expression with respect to the variable is computed:

```
>> interactiveDiff :=
     proc()
       local f, x;
     begin
        f := input("enter an expression: ");
        x := input("enter an identifier: ");
        print(Unquoted, "The derivative of " . expr2text(f) .
              " with respect to ". expr2text(x) . " is:");
        diff(f, x)
     end_proc:

>> interactiveDiff()

 enter an expression: << x^2 + x*y^3 >>
 enter an identifier: << x >>

       The derivative of x^2 + x*y^3 with respect to x is:

                                   3
                            2 x + y
```

The function `input` does not evaluate the input. This leads to the following unexpected result:

```
>> f := x^2 + x*y^3:
   z := x:
   interactiveDiff()
```

```
 enter an expression: << f >>
 enter an identifier: << z >>

            The derivative of f with respect to z is:

                                    0
```

The following modification enforces full evaluation via `eval`:

```
>> interactiveDiff :=
     proc()
       local f, x;
     begin
        f := eval(input("enter an expression: "));
        x := eval(input("enter an identifier: "));
        print(Unquoted, "The derivative of " . expr2text(f) .
              " with respect to ". expr2text(x) . " is:");
        diff(f, x)
     end_proc:

>> interactiveDiff()

 enter an expression: << f >>
 enter an identifier: << z >>

       The derivative of x^2 + x*y^3 with respect to x is:

                                       3
                               2 x + y

>> delete interactiveDiff, f, z:
```

`int` – definite and indefinite integration

`int(f, x)` computes the indefinite (formal) integral $\int f(x)\,\mathrm{d}x$.

`int(f, x = a..b)` computes the definite integral $\int_a^b f(x)\,\mathrm{d}x$.

Call(s):

- `int(f, x)`
- `int(f, x = a..b` <, *Continuous*>`)`
- `int(f, x = a..b` <, *PrincipalValue*>`)`

Parameters:

`f`	—	the integrand: an arithmetical expression representing a function in `x`
`x`	—	the integration variable: an identifier
`a, b`	—	the boundaries: arithmetical expressions

Options:

Continuous	—	do not look for discontinuities.
PrincipalValue	—	compute the Cauchy principal value of the integral.

Return Value: an arithmetical expression.

Overloadable by: `f`

Side Effects: `int` is sensitive to properties of identifiers set by `assume`; see example 4.

Further Documentation: Section 7.2 of the MuPAD Tutorial.

Related Functions: `D`, `diff`, `intlib`, `limit`, `numeric::int`, `sum`

Details:

- `int(f, x)` computes the antiderivative $g = \int f \, \mathrm{d}x$, i.e., it determines formally a function g with $\partial g / \partial x = f$. Note:

  - No constant of integration appears in the result.
  - The result is not necessarily continuous, even if the integrand is continuous. See "background" section for more details.
  - In general, the derivative of the result coincides with f only on some open interval of the real domain.

  It is not always possible to decide algorithmically whether $\partial g / \partial x$ and f are equivalent. This is due to the so-called zero equivalence problem, which in general is undecidable.

- For the case of indefinite integration, the integration variable `x` is implicitly assumed to be real. For definite integration the integration variable `x` is further implicitly assumed to be restricted to the given real range of integration. See "background" section for more details.

  This means that in general, the result of `int` need not be valid for non-real values of `x`, e.g., the identity $\ln(\exp(x)) = x$ is only valid for real values of x and thus the same is true for $\int \ln(\exp(x)) \, \mathrm{d}x = x^2/2$.

- If MuPAD cannot find a closed form solution for the integral, then it returns a symbolic `int` call. In this case, you can use numerical integration (cf. example 2) or try to compute a series expansion of the integral (cf. example 3).
- For definite integrals, `int` may not be able to find a closed form due to singularities in the interval of integration. If the system can assert that the integral does not exist mathematically, then it returns `undefined`. In some cases, it may still be possible to obtain a result in closed form by using assumptions or one of the options *Continuous* or *PrincipalValue* (cf. example 4).
- Numerical approximations to a definite integral can be obtained with `numeric::int` or `float`. Numerical integration is only possible if the boundaries `a` and `b` can be converted into floating point numbers via `float`. See example 2.

Option <`Continuous`>**:**

- For definite integration, the system may first compute an antiderivative `g` of `f` with respect to `x`, such that $\partial g/\partial x = f$. If `g` is continuous on the interval $[a, b]$, then the fundamental theorem of calculus $\int_a^b f(x)\,\mathrm{d}x = g(a) - g(b)$ is used to obtain the definite integral. Normally, it is tested if `g` is continuous. In case of doubt a symbolic `int` call is returned. See "background" section for more details.

  The option *Continuous* is a *technical* option to tell the system that it may assume that `g` is continuous. With the option *Continuous*, `int` suppresses the search for discontinuities of `g` in the interval of integration and uses the fundamental theorem of calculus without checking whether it applies mathematically. See example 4.

Option <`PrincipalValue`>**:**

- If the interior of the interval of integration contains poles of the integrand or the boundaries are $a = -\infty$ and $b = \infty$, then the definite integral may not exist in a strict mathematical sense. However, if the integrand changes sign at all poles in the interval of integration, then a weaker form of definite integral, the *Cauchy principal value*, which allows "infinite parts" of the integral to the left and to the right of a pole to cancel each other, may still exist. With the option *PrincipalValue*, `int` computes this Cauchy principal value. If the usual definite integral exists, then it agrees with the Cauchy principal value. See example 4.

Example 1. We compute the two indefinite integrals $\int \frac{1}{x \ln x}\,\mathrm{d}x$ and $\int \frac{1}{x^2-8}\,\mathrm{d}x$:

```
>> int(1/x/ln(x), x)

                        ln(ln(x))

>> int(1/(x^2 - 8), x)

          1/2           1/2     1/2           1/2
         2    ln(x - 2 2   )   2    ln(x + 2 2   )
         ------------------- - -------------------
                  8                     8
```

We compute the definite integral of $(x\ln(x))^{-1}$ over the interval $[e, e^2]$:

```
>> int(1/x/ln(x), x = exp(1)..exp(2))

                          ln(2)
```

The boundaries of definite integrals may be $\pm\infty$ as well:

```
>> int(exp(-x^2), x = 0..infinity)

                            1/2
                          PI
                          -----
                            2
```

One can also determine multiple integrals such as, e.g., the definite multiple integral $\int_0^a \int_0^{1-x/a} \int_0^{1-x/a-y/b} \mathrm{d}z\,\mathrm{d}y\,\mathrm{d}x$:

```
>> int(int(int(1, z = 0..c*(1 - x/a - y/b)),
           y = 0..b*(1 - x/a)), x = 0..a)

                          a b c
                          -----
                            6
```

Example 2. The system cannot find a closed form for the following definite integral and returns a symbolic `int` call. You can obtain a floating point approximation by applying `float` to the result:

```
>> int(sin(cos(x)), x = 0..1)

                   int(sin(cos(x)), x = 0..1)

>> float(%)

                       0.738642998
```

Alternatively, you can use the function `numeric::int`. This is recommended if you are interested only in a numerical approximation, since it does not involve any symbolic preprocessing and is therefore usually much faster than applying `float` to a symbolic `int` call.

```
>> numeric::int(sin(cos(x)), x = 0..1)

                        0.738642998
```

Example 3. `int` cannot find a closed form for the following indefinite integral and returns a symbolic `int` call:

```
>> int((x^2 + 1)/sqrt(x^3 + 1), x)

                     /      2           \
                     |     x  + 1       |
                  int| -----------, x |
                     |    3     1/2     |
                     \ (x  + 1)         /
```

You can use `series` to obtain a series expansion of the integral:

```
>> series(%, x = 0)

                      3    4    6
                     x    x    x        7
                 x + -- - -- - -- + O(x )
                      3    8   12
```

Alternatively, you can compute a series expansion of the integrand and integrate it afterwards. This is recommended if you are not interested in a closed form of the integral, but only in a series expansion, since it is usually much faster than the other way round:

```
>> int(series((x^2 + 1)/sqrt(x^3 + 1), x = 0), x)

                      3    4    6
                     x    x    x        7
                 x + -- - -- - -- + O(x )
                      3    8   12
```

Example 4. `int` correctly asserts that the following definite integral, where the integrand has a pole in the interior of the interval of integration, is not defined:

```
>> int(1/(x - 1), x = 0..2)
```

```
undefined
```

However, the Cauchy principle value of the integral exists:

```
>> int(1/(x - 1), x = 0..2, PrincipalValue)

                                0
```

If, however, the integrand contains a parameter, then `int` may not be able to decide whether the integrand has poles in the interval of integration. In such a case, a warning is issued and a symbolic `int` call is returned:

```
>> int(1/(x - a), x = 0..2)

 Warning: Found potential discontinuities of the antiderivative\
 .
 Try option 'Continuous' or use properties (?assume). [intlib::\
 antiderivative]

                         /   1            \
                      int| -----, x = 0..2 |
                         \ x - a          /
```

We follow the suggestion given by the text of the warning and make an assumption on the parameter `a` implying that the integrand has no poles in the interval of integration. In this example, `int` is able to find a closed form of the integral:

```
>> assume(a > 2): int(1/(x - a), x = 0..2)

                        ln(2 - a) - ln(-a)
```

Alternatively, we can use the option *Continuous* to tell `int` that it may assume that the integrand is continuous in the range of integration:

```
>> unassume(a): int(1/(x - a), x = 0..2, Continuous)

                        ln(2 - a) - ln(-a)
```

Mathematically, the result with option *Continuous* may be incorrect for some values of the occurring parameters. In the example above, the result is incorrect for $0 < a < 2$. We therefore recommend to use this option only as a last resort.

Example 5. In this example we will stress the effects of assumptions on the integration variable. See "background" section for more details.

The integration variable is implicitly assumed to be real, or even for a given (real) interval of integration restricted to that interval. Among other things this assumption has an impact on the simplification of results.

For example, to compute the following integral internally the so-called Risch algorithm is used and only because of that implicit assumption the result is simplified into a real representation.

```
>> int(1/cos(x)^2, x)

                 2 sin(2 x)
               --------------
               2 cos(2 x) + 2
```

In order to see what will happen without this implicit assumption one can explicitly define the integration variable to be complex:

```
>> assume(x, Type::Complex): int(1/cos(x)^2, x)

                          2 I
         - -------------------------
             cos(2 x) - I sin(2 x) + 1
```

User-defined assumptions which are inconsistent with the assumptions made internally in the integration do not lead to an integration error as they should. However, the user must become aware of the inconsistency.

```
>> assume(x, Type::Imaginary): int(1/cos(x)^2, x)

 Warning: Cannot integrate when x has property Type::Imaginary.
 While integrating, we will assume x has property Type::Complex\
 . [intlib::int]

                          2 I
         - -------------------------
             cos(2 x) - I sin(2 x) + 1

>> assume(x, Type::Integer): int(1/cos(x)^2, x)

 Warning: Cannot integrate when x has property Type::Integer.
 While integrating, we will assume x has property Type::Real. [\
 intlib::int]

                 2 sin(2 x)
               --------------
               2 cos(2 x) + 2
```

The same holds for definite integration.

```
>> assume(x, Type::Interval(-5, -2)): int(x, x = 0..1)

 Warning: While integrating, we will assume x has property [0, \
 1] instead of given property ]-5, -2[. [intlib::defInt]

                        1/2
```

Background:

- With the integration techniques used in computer algebra like table lookup or Risch integration for an indefinite integral, in addition to the possible discontinuities of the initial integrand, some more discontinuities may occur during the integration process. This is due to the fact that algebraic numbers can be complex. It may cause branch problems in numerical computation, since, e.g., the arguments to the logarithms may have complex zeros while the initial integrand has no pole in the path of integration. If the classical algorithm is used for rewriting complex logarithms as real-valued arcus-tangents,

  $$\sqrt{-1}\frac{\mathrm{d}}{\mathrm{d}x}\ln\left(\frac{u+\sqrt{-1}}{u-\sqrt{-1}}\right) = 2\frac{\mathrm{d}}{\mathrm{d}x}\arctan(u)$$

  where u is an element of $K(x)$ such that $u^2 \neq -1$ and K is a subfield of the reals, it does not eliminate the problem. However, it may be used in some integration tables.

  Thus, if such results are used for definite integration, it is necessary to investigate the search for discontinuities of the antiderivatives in the interval of integration.

- The integration variable is implicitly assumed to be real (`Type::Real`). Moreover, for definite integration, the range of validity is restricted to the interval of integration (`Type::Interval[a, b]`).

  If conflicts occur with user-defined properties by `assume` of identifiers, an appropriate warning is given. The warnings may be toggled on and off with `intlib::printWarnings(TRUE)` and `intlib::printWarnings(FALSE)`.

  In the case of indefinite integration the user-defined properties are used if the conflict can be resolved. If not, but the given properties describe a subset of the real numbers, the real assumption is used. Otherwise, while integrating, the integration variable is assumed to be complex.

  If, in the case of definite integration, the user-defined properties contains the given integration interval, these properties are used. Otherwise, the previously given assumption are set locally.

  Cf. example 5.

- For details of the algorithms and simplification strategies see:

  - M. Bronstein. A Unification of Liouvillian Extension. AAECC Applicable Algebra in Engineering, Communication and Computing. 1: 5–24, 1990.

  - M. Bronstein. The Transcendental Risch Differential Equation. Journal of Symbolic Computation. 9: 49–60, 1990.

- M. Bronstein. Symbolic Integration I: Transcendental Functions. Springer. 1997.
- H. I. Epstein and B. F. Caviness. A Structure Theorem for the Elementary Functions and its Application to the Identity Problem. International Journal of Computer and Information Science. 8: 9–37, 1979.
- W. Fakler. Vereinfachen von komplexen Integralen reeller Funktionen. mathPAD 9 No. 1: 5-9, 1999.
- K. O. Geddes, S. R. Czapor and G. Labahn. Algorithms for Computer Algebra. 1992.

`int2text` – convert an integer to a character string

`int2text(n, b)` converts the integer `n` to a string that corresponds to the `b`-adic representation of `n`.

Call(s):

- `int2text(n <, b>)`

Parameters:

`n` — an integer
`b` — the base: an integer between 2 and 36. The default base is 10.

Return Value: a character string.

Related Functions: `coerce`, `expr2text`, `genpoly`, `numlib::g_adic`, `tbl2text`, `text2expr`, `text2int`, `text2list`, `text2tbl`

Details:

- The string returned by `int2text` consists of the first `b` characters in

  $$0, 1, \ldots, 9, A, B, \ldots, Z.$$

  For bases larger than 10, the letters represent the `b`-adic digits larger than 9: $A = 10, B = 11, \ldots, Z = 35$.

- For the bases 2, 8, or 16, `int2text` provides the conversion from decimal representation to binary, octal, or hexadecimal representation, respectively.

- `int2text` is the inverse of `text2int`.

- Since the output of the numerical datatypes in MuPAD uses the decimal representation, strings are used by `int2text` to represent `b`-adic numbers. The function `numlib::g_adic` provides an alternative representation via lists.
- `int2text` is a function of the system kernel.

Example 1. Relative to the default base 10, `int2text` provides a mere datatype conversion from `DOM_INT` to `DOM_STRING`:

```
>> int2text(123), int2text(-45678)

                        "123", "-45678"
```

Example 2. The decimal integer 32 has the following binary representation:

```
>> int2text(32, 2)

                          "100000"
```

The decimal integer 10^9 has the following hexadecimal representation:

```
>> int2text(10^9, 16)

                         "3B9ACA00"
```

Example 3. Negative integers can be converted as well:

```
>> int2text(-15, 8)

                           "-17"
```

`interval` – convert constant subexpressions to intervals

`interval(object)` converts all constant subexpressions of `object` to floating point intervals.

Call(s):

- `interval(object)`

Parameters:

`object` — an arbitrary MuPAD object

Return Value: a MuPAD object

Related Functions: `Dom::FloatIV`, `float`, `hull`, `misc::maprec`

Details:

- `interval` is the analogue of `float`. While the latter converts exact numbers and numerical expressions to floating point approximations, `interval` converts numbers and numerical expressions to enclosing floating point intervals.
- If `object` is an arithmetical expression, `interval(object)` recursively descends into the subexpressions of `object` and replaces all integers, rationals, and floating point numbers as well as the constant `PI` by floating point intervals enclosing them. Afterwards, the resulting expression is evaluated via interval arithmetic.
- If `object` is not an arithmetical expression, `interval` returns the object unchanged.
- `interval` is a function of the system kernel.

Example 1. Only constant expressions such as numbers 1, 2/3, $0.123 + 4.5 * I$ etc. and numerical expressions `PI + sqrt(2)`, `sin(PI/24)` etc. are converted to floating point intervals. Symbolic objects such as identifiers, indexed identifiers etc. are left untouched:

```
>> interval(4*x[1] + PI*x[2]^2/sin(1) + 1/4)

                                                     2
 (4.0 ... 4.0) x[1] + (3.733453333 ... 3.733453334) x[2]  +

    (0.25 ... 0.25)

>> interval(f(g(2 + x) + sin(1)*sqrt(PI)))

     f(g(x + (2.0 ... 2.0)) + (1.491468487 ... 1.491468488))
```

Example 2. Of the special MuPAD constants `CATALAN`, `EULER` and `PI`, presently only `PI` can be converted to an enclosing floating point interval. The other constants are left untouched:

```
>> interval(CATALAN), interval(EULER), interval(PI)

           CATALAN, EULER, 3.141592653 ... 3.141592654
```

Changes:

- `interval` is a new function.

`irreducible` – test irreducibility of a polynomial

`irreducible(p)` tests if the polynomial `p` is irreducible.

Call(s):

- `irreducible(p)`

Parameters:

`p` — a polynomial of type `DOM_POLY` or a polynomial expression

Return Value: `TRUE` or `FALSE`.

Overloadable by: `p`

Related Functions: `content`, `factor`, `gcd`, `icontent`, `ifactor`, `igcd`, `ilcm`, `isprime`, `lcm`, `poly`, `polylib::divisors`, `polylib::primpart`, `polylib::sqrfree`

Details:

- A polynomial $p \in k[x_1, \ldots, x_n]$ is irreducible over the field k if p is nonconstant and is not a product of two nonconstant polynomials in $k[x_1, \ldots, x_n]$.
- `irreducible` returns `TRUE` if the polynomial is irreducible over the field implied by its coefficients. Otherwise, `FALSE` is returned. See the function `factor` for details on the coefficient field that is assumed implicitly.
- The polynomial may be either a (multivariate) polynomial over the rationals, a (multivariate) polynomial over a field (such as the residue class ring `IntMod(n)` with a prime number `n`) or a univariate polynomial over an algebraic extension (see `Dom::AlgebraicExtension`).
- Internally, a polynomial expression is converted to a polynomial of type `DOM_POLY` before irreducibility is tested.

Example 1. With the following call, we test if the polynomial expression $x^2 - 2$ is irreducible. Implicitly, the coefficient field is assumed to consist of the rational numbers:

```
>> irreducible(x^2 - 2)

                              TRUE

>> factor(x^2 - 2)

                               2
                              x  - 2
```

Since $x^2 - 2$ factors over a field extension of the rationals containing the radical $\sqrt{2}$, the following irreducibility test is negative:

```
>> irreducible(sqrt(2)*(x^2 - 2))

                              FALSE

>> factor(sqrt(2)*(x^2 - 2))

                     1/2       1/2        1/2
                    2    (x + 2   ) (x - 2   )
```

The following calls use polynomials of type `DOM_POLY`. The coefficient field is given explicitly by the polynomials:

```
>> irreducible(poly(6*x^3 + 4*x^2 + 2*x - 4, IntMod(13)))

                              TRUE

>> factor(poly(6*x^3 + 4*x^2 + 2*x - 4, IntMod(13)))

                   3      2
         6 poly(x  + 5 x  - 4 x - 5, [x], IntMod(13))

>> irreducible(poly(3*x^2 + 5*x + 2, IntMod(13)))

                              FALSE

>> factor(poly(3*x^2 + 5*x + 2, IntMod(13)))

   3 poly(x + 5, [x], IntMod(13)) poly(x + 1, [x], IntMod(13))
```

`is` – check a mathematical property of an expression

`is(x, prop)` checks whether the expression `x` has the mathematical property `prop`.

`is(y rel z)` checks whether the relation `rel` holds for the expressions `y` and `z`.

`is(x in set)` checks whether `x` is an element of the set.

Call(s):

- `is(x, prop)`
- `is(y rel z)`
- `is(x in set)`

Parameters:

`x, y, z`	—	arithmetical expressions
`prop`	—	a property
`rel`	—	one of `=`, `<`, `>`, `<=`, `>=`, `<>`
`set`	—	a property representing a set of numbers (e.g., `Type::PosInt`) or a set returned by `solve`; such a set can be an element of `Dom::Interval`, `Dom::ImageSet`, `piecewise`, or one of `C_`, `R_`, `Q_`, `Z_`.

Return Value: `TRUE`, `FALSE`, or `UNKNOWN`.

Related Functions: `assume`, `bool`, `getprop`, `property::implies`, `unassume`

Details:

- The property mechanism helps to simplify expressions involving identifiers that carry "mathematical properties". The function `assume` allows to attach basic properties ("assumptions") such as '`x` is a real number' or '`x` is an odd integer' to an identifier `x`, say. Arithmetical expressions involving `x` may inherit such properties. E.g., '`1 + x^2` is positive' if '`x` is a real number'. The function `is` is the basic tool for querying mathematical properties.

  See the `property` library for a description of all available properties.

- `is` queries the properties of the given expressions via `getprop`. Then it checks whether the property `prop` or the relation `y rel z` can be derived from the properties of `x`, `y`, and `z`. If this is the case, then `is` returns `TRUE`. If `is` derives the logical negation of the property `prop` or the relation `y rel z`, respectively, then it returns `FALSE`. Otherwise, `is` returns `UNKNOWN`.

- If a relation is given to `is`, and the operands are complex numbers or identifiers with this property, `is` returns `FALSE`, because a relations holds only with real objects. Cf. example 4.

- It may happen that `is` returns `UNKNOWN`, although the queried property holds mathematically. Cf. example 5.

- In MuPAD, there also exists the function `bool` to check a relation `y rel z`. However, there are two main differences between `bool` and `is`:

1. `bool` produces an error if it cannot decide whether the relation holds or not; `is(y rel z)` returns `UNKNOWN` in this case.

2. `bool` does not take properties into account.

Cf. example 3.

- If `bool(y rel z)` returns `TRUE`, then so does `is(y rel z)`. However, `is` is more powerful than `bool`, even when no properties are involved. Cf. example 3. On the other hand, `is` is usually much slower than `bool`.

- Be careful when using `is` in a condition of an `if` statement or a `for`, `while`, or `repeat` loop: these constructs cannot handle the value `UNKNOWN`. Use either `is(...) = TRUE` or a `case` statement. Cf. example 6.

- If `is` needs to check whether a constant symbolic expression is zero, then it may employ a heuristic numerical zero test based on floating point evaluation. Despite internal numerical stabilization, this zero test may return the wrong answer in exceptional pathological cases; in such a case, `is` may return a wrong result as well.

Example 1. The identifier `x` is assumed to be an integer:

```
>> assume(x, Type::Integer):
   is(x, Type::Integer), is(x > 0), is(x^2 >= 0)

                    TRUE, UNKNOWN, TRUE
```

The identifier `x` is assumed to be a positive real number:

```
>> assume(x > 0): is(x > 1), is(x >= 0), is(x < 0)

                    UNKNOWN, TRUE, FALSE

>> unassume(x):
```

Example 2. `is` can derive certain facts even when no properties were assumed explicitly:

```
>> is(x > x + 1), is(abs(x) >= 0)

                    FALSE, TRUE

>> is(Re(exp(x)), Type::Real)

                    TRUE
```

Example 3. For relations between numbers, `is` yields the same answers as `bool`:

```
>> bool(1 > 0), is(1 > 0)
```

```
TRUE, TRUE
```

However, on constant symbolic expressions, `is` can realize more than `bool`:

```
>> is(sin(5) > 1/2), is(PI^3 + 2 < 33), is(exp(1) > exp(0.9))
```

```
FALSE, FALSE, TRUE
```

```
>> bool(sin(5) > 1/2)
```

```
Error: Can't evaluate to boolean [_less]
```

```
>> is(sqrt(2) > 1.4), is(PI > 3.1415)
```

```
TRUE, TRUE
```

```
>> bool(sqrt(2) > 1.4)
```

```
Error: Can't evaluate to boolean [_less]
```

```
>> is(exp(5), Type::Real), is(PI, Type::PosInt)
```

```
TRUE, FALSE
```

Example 4. In the next example a relation with complex objects is given, the returned value is `FALSE`:

```
>> is(0 < I), is(I + 1 > I), is(1 + 2*I <= 2 + 3*I)
```

```
FALSE, FALSE, FALSE
```

The identifier in the next example is assumed to be complex, but it could be real too:

```
>> assume(x, Type::Complex):
   is(x > 0)
```

```
UNKNOWN
```

With the following assumption the identifier `x` cannot be real, therefore `is` returns `FALSE`:

```
>> assume(x, not Type::Real):
   is(x >= 0)
```

```
FALSE
```

The next relation is false, either the identifier `x` is real, then the relation is false, or the identifiers is not real, then the comparison is illegal:

```
>> unassume(x):
   is(x + 1 < x)

FALSE

>> unassume(x):
```

Example 5. Here are some examples where the queried property can be derived mathematically. However, the current implementation of `is` is not yet strong enough to derive the property:

```
>> assume(x, Type::Real): is(abs(x) >= x)

UNKNOWN

>> assume(x, Type::Interval(0, PI)): is(sin(x) >= 0)

UNKNOWN

>> unassume(x):
```

Example 6. Care must be taken when using `is` in `if` statements or `for`, `repeat`, `while` loops:

```
>> myabs := proc(x)
            begin
              if is(x >= 0) then
                x
              elif is(x < 0) then
                -x
              else
                procname(x)
              end_if
            end_proc:

>> assume(x < 0): myabs(1), myabs(-2), myabs(x)

1, 2, -x
```

When the call of `is` returns `UNKNOWN`, an error occurs because `if` expects `TRUE` or `FALSE`:

```
>> unassume(x): myabs(x)

 Error: Can't evaluate to boolean [if];
 during evaluation of 'myabs'
```

The easiest way to achieve the desired functionality is a comparison of the result of `is` with `TRUE`:

```
>> myabs := proc(x)
            begin
              if is(x >= 0) = TRUE then
                x
              elif is(x < 0) = TRUE then
                -x
              else
                procname(x)
              end_if
            end_proc:

>> myabs(x)

                            myabs(x)

>> delete myabs:
```

Example 7. `is` can handle sets returned by `solve`. These include intervals of type `Dom::Interval` and `R_` = `solvelib::BasicSet(Dom::Real)`:

```
>> assume(x >= 0): assume(x <= 1, _and):
   is(x in Dom::Interval([0, 1])), is(x in R_)

                           TRUE, TRUE
```

The following `solve` command returns the solution as an infinite parameterized set of type `Dom::ImageSet`:

```
>> unassume(x): solutionset := solve(sin(x) = 0, x)

                      { X1*PI |  X1 in Z_ }

>> domtype(solutionset)

                          Dom::ImageSet
```

`is` can be used to check whether an expression is contained in this set:

```
>> is(20*PI in solutionset), is(PI/2 in solutionset)

                           TRUE, FALSE
```

```
>> delete solutionset:
```

---

isprime – primality test

isprime(n) checks whether n is a prime number.

Call(s):

- isprime(n)

Parameters:

n — an arithmetical expression representing an integer

Return Value: either TRUE or FALSE, or a symbolic isprime call.

Related Functions: factor, ifactor, igcd, ilcm, irreducible, ithprime, nextprime, numlib::primedivisors, numlib::prevprime, numlib::proveprime

---

Details:

- isprime is a fast probabilistic prime number test (Miller-Rabin test). The function returns TRUE when the positive integer n is either a prime number or a strong pseudo-prime for 10 independently and randomly chosen bases. Otherwise, isprime returns FALSE.
- If n is positive and isprime returns FALSE, then n is guaranteed to be composite. If n is positive and isprime returns TRUE, then n is prime with a very high probability.

  Use numlib::proveprime for a prime number test that always returns the correct answer. Note, however, that it is usually much slower than isprime.
- isprime(0) and isprime(1) return FALSE. isprime returns always FALSE if n is a negative integer.
- isprime returns an error message if its argument is a number but not an integer. isprime returns a symbolic isprime call if the argument is not a number.
- isprime is a function of the system kernel.

Example 1. The number `989999` is prime:

```
>> isprime(989999)
```

```
TRUE
```

```
>> ifactor(989999)
```

```
989999
```

In contrast to `ifactor`, `isprime` can handle large numbers:

```
>> isprime(2^(2^11) + 1)
```

```
FALSE
```

`isprime(0)` and `isprime(1)` return `FALSE`:

```
>> isprime(0), isprime(1)
```

```
FALSE, FALSE
```

Negative numbers yield `FALSE` as well:

```
>> isprime(-13)
```

```
FALSE
```

For non-numeric arguments, a symbolic `isprime` call is returned:

```
>> delete n: isprime(n)
```

```
isprime(n)
```

Background:

- Reference: Michael O. Rabin, Probabilistic algorithms, in J. F. Traub, ed., *Algorithms and Complexity*, Academic Press, New York, 1976, pp. 21–39.

`isqrt` – integer square root

`isqrt(n)` computes an integer approximation to the square root of the integer `n`.

Call(s):

- `isqrt(n)`

Parameters:

n — an arithmetical expression representing an integer

Return Value: a nonnegative integer, an integral multiple of I, or a symbolic isqrt call.

Overloadable by: n

Related Functions: _power, icontent, ifactor, igcd, ilcm, numlib::ispower, numlib::issqr, sqrt, trunc

---

Details:

- If n is a perfect square, then isqrt returns the unique nonnegative integer whose square is n. More generally, if n is a nonnegative integer, then isqrt computes trunc(sqrt(n)). Thus the approximation error is less than 1.
- If n is a negative integer, then isqrt computes trunc(sqrt(-n))*I.
- isqrt returns an error message if its argument is a number but not an integer. isqrt returns a symbolic isqrt call if the argument is not a number.
- isqrt is a function of the system kernel.

Example 1. We compute some integer square roots:

```
>> isqrt(4), isqrt(5)

                    2, 2
```

The approximation error is less than 1:

```
>> isqrt(99), float(sqrt(99))

               9, 9.949874371
```

The integer square root of a negative integer is an integral multiple of I:

```
>> isqrt(-4), isqrt(-5)

                 2 I, 2 I
```

If the argument is not a number, the result is a symbolic isqrt call:

```
>> delete n: isqrt(n)

                 isqrt(n)
```

```
>> type(%)
```

"isqrt"

`iszero` – generic zero test

`iszero(object)` checks whether `object` is the zero element in the domain of `object`.

Call(s):

- `iszero(object)`

Parameters:

`object` — an arbitrary MuPAD object

Return Value: either `TRUE` or `FALSE`

Overloadable by: `object`

Related Functions: `_equal`, `Ax::normalRep`, `bool`, `is`, `normal`, `simplify`, `sign`

Details:

- Use the condition `iszero(object)` instead of `object = 0` to decide whether `object` is the zero element, because `iszero(object)` is more general than `object = 0`. If the call `bool(object = 0)` returns `TRUE`, then `iszero(object)` returns `TRUE` as well, but in general not vice versa (see example 1).
- If `object` is an element of a basic type, then `iszero` returns `TRUE` precisely if one of the following is true: `object` is the integer `0` (of domain type `DOM_INT`), the floating point value `0.0` (of domain type `DOM_FLOAT`), the floating point interval (of domain type `DOM_INTERVAL`) `0...0`, or the zero polynomial (of domain type `DOM_POLY`). In the case of a polynomial, the result `FALSE` is guaranteed to be correct only if the coefficients of the polynomial are in normal form (i.e., if zero has a unique representation in the coefficient ring). See also `Ax::normalRep`.
- If `object` is an element of a library domain, then the method `"iszero"` of the domain is called and the result is returned. If this method does not exist, then the function `iszero` returns `FALSE`.

- `iszero` performs a purely syntactical zero test. If `iszero` returns `TRUE`, then the answer is always correct. If `iszero` returns `FALSE`, however, then it may still be true that mathematically `object` represents zero (see example 3). In such cases, the MuPAD functions `normal` or `simplify` may be able to recognize this.

- `iszero` does *not* take into account properties of identifiers in `object` that have been set via `assume`. In particular, you should not use `iszero` in an argument passed to `assume` or `is`; use the form `object = 0` instead (see example 2).

- Do not use `iszero` in a condition passed to `piecewise`. In contrast to `object = 0`, the command `iszero(object)` is evaluated immediately, before it is passed to `piecewise`, while the evaluation of `object = 0` is handled by `piecewise` itself. Thus using `iszero` in a `piecewise` command usually leads to unwanted effects (see example 4).

- `iszero` is a function of the system kernel.

Example 1. `iszero` handles the basic data types:

```
>> iszero(0), iszero(1/2), iszero(0.0), iszero(I), iszero(-1...1)

                  TRUE, FALSE, TRUE, FALSE, FALSE
```

`iszero` works for polynomials:

```
>> p:= poly(x^2 + y, [x]):
   iszero(p)

                                FALSE

>> iszero(poly(0, [x, y]))

                                TRUE
```

`iszero` is more general than `=`:

```
>> bool(0 = 0), bool(0.0 = 0), bool(poly(0, [x]) = 0)

                          TRUE, FALSE, FALSE

>> iszero(0), iszero(0.0), iszero(poly(0, [x]))

                           TRUE, TRUE, TRUE
```

Example 2. `iszero` does not react to properties:

```
>> assume(a = b): is(a - b = 0)
```

```
TRUE
```

```
>> iszero(a - b)
```

```
FALSE
```

Example 3. Although `iszero` returns `FALSE` in the following example, the expression in question mathematically represents zero:

```
>> iszero(sin(x)^2 + cos(x)^2 - 1)
```

```
FALSE
```

In this case `simplify` is able to decide this:

```
>> simplify(sin(x)^2 + cos(x)^2 - 1)
```

```
0
```

Example 4. `iszero` should not be used in a condition passed to `piecewise`:

```
>> delete x:
   piecewise([iszero(x), 0], [x <> 0, 1])
```

```
piecewise(1 if x <> 0)
```

The first branch was discarded because `iszero(x)` immediately evaluates to `FALSE`. Instead, use the condition `x = 0`, which is passed unevaluated to `piecewise`:

```
>> piecewise([x = 0, 0], [x <> 0, 1])
```

```
piecewise(0 if x = 0, 1 if x <> 0)
```

`ithprime` – the i-th prime number

`ithprime(i)` returns the i-th prime number.

Call(s):

- `ithprime(i)`

Parameters:

i — an arithmetical expression

Return Value: a prime number or an unevaluated call to `ithprime`

Related Functions: `ifactor`, `igcd`, `ilcm`, `isprime`, `nextprime`, `numlib::prevprime`

Details:

- If the argument `i` is a positive integer, then `ithprime` returns the i-th prime number. An unevaluated call is returned, if the argument is not of type `Type::Numeric`. An error occurs if the argument is a number that is not a positive integer.
- The first prime number `ithprime(1)` is 2.
- If the i-th prime number is contained in the system's internal prime number table (see the help page for `ifactor`), then it is returned by a fast kernel function. Otherwise, MuPAD iteratively calls `nextprime`, using some suitable pre-computed value of `ithprime` as starting point. This is still reasonably fast for $i \leq 1000000$. If i exceeds this value, however, then the run time grows exponentially with the number of digits of i.

Example 1. The first 10 prime numbers:

```
>> ithprime(i) $ i = 1..10

               2, 3, 5, 7, 11, 13, 17, 19, 23, 29
```

A larger prime:

```
>> ithprime(123456)

                              1632899
```

Symbolic arguments lead to an unevaluated call:

```
>> ithprime(i)

                            ithprime(i)
```

`lambertV, lambertW` – lower and upper real branch of the Lambert function

For real x, the values $y = \texttt{lambertV}(x)$ and $y = \texttt{lambertW}(x)$ represent the real solutions of the equation $y\, e^y = x$.

Call(s):

- `lambertV(x)`
- `lambertW(x)`

Parameters:

`x` — an arithmetical expression

Return Value: an arithmetical expression.

Side Effects: When called with a floating point argument, the functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Details:

- For all real $x \geq 0$, the equation $y\, e^y = x$ has exactly one real solution. It is represented by $y = \texttt{lambertW}(x)$.

  For all real x in the range $0 > x > -e^{-1}$, there are exactly two real solutions. The larger one is represented by $y = \texttt{lambertW}(x)$, the smaller one by $y = \texttt{lambertV}(x)$.

  Exactly one real solution $\texttt{lambertW}(-e^{-1}) = \texttt{lambertV}(-e^{-1}) = -1$ exists for $x = -e^{-1}$.

- Thus, the upper branch `lambertW` is defined for real arguments from the interval $[-e^{-1}, \infty)$. It is monotonically increasing, attaining values in the interval $[-1, \infty)$.

  The lower branch `lambertV` is defined for real arguments from the interval $[-e^{-1}, 0)$. It is monotonically decreasing, attaining values in the interval $[-1, -\infty)$.

- The values $\texttt{lambertV}(0) = -\texttt{infinity}$ and $\texttt{lambertW}(0) = 0$ are implemented. Further, the result y is returned for some exact arguments of the form $x = y\, e^y$. For real floating point arguments from the range of definition a floating point value is returned. For all other arguments, unevaluated function calls are returned.

- The `float` attributes are kernel functions, i.e., floating point evaluation is fast.

Example 1. We demonstrate some calls with exact and symbolic input data:

```
>> lambertV(-4), lambertW(-3), lambertV(-5/2), lambertW(1/2),
   lambertV(I), lambertW(1 + I), lambertV(x + 1)
```

```
 lambertV(-4), lambertW(-3), lambertV(-5/2), lambertW(1/2),

    lambertV(I), lambertW(1 + I), lambertV(x + 1)
```

Some exact values are found:

```
>> lambertV(-exp(-1)), lambertW(-2*exp(-2)),
   lambertV(-3/2*exp(-3/2)), lambertW(exp(1)),
   lambertW(2*exp(2)), lambertW(5/2*exp(5/2))

                     -1, -2, -3/2, 1, 2, 5/2
```

Floating point values are computed for floating point arguments:

```
>> lambertV(-0.3), lambertW(2000.0)

                    -1.781337023, 5.836731495
```

The following arguments are not from the range of definition and lead to unevaluated calls:

```
>> lambertV(-1.0), lambertW(-0.4), lambertV(0.1),
   lambertV(exp(1)), lambertV(5*exp(5))

 lambertV(-1.0), lambertW(-0.4), lambertV(0.1),

    lambertV(exp(1)), lambertV(5 exp(5))
```

Example 2. The functions `diff`, `float`, and `series` handle expressions involving the Lambert function:

```
>> diff(lambertV(x), x), diff(lambertW(x), x)

                lambertV(x)          lambertW(x)
            -------------------, -------------------
            x (lambertV(x) + 1)  x (lambertW(x) + 1)

>> float(ln(3 + lambertW(sqrt(PI))))

                           1.334475971

>> series(lambertW(x), x = 0);
   series(lambertW(x), x = -1/exp(1), 3);
   series(lambertV(x), x = -1/exp(1), 3);
```

```
                   3      4        5       6
            2   3 x    8 x    125 x    54 x       7
       x - x  + ---- - ---- + ------ - ----- + O(x )
                  2      3      24       5

       1/2       1/2 /        1   \1/2
- 1 + 2    exp(1)    | x + ------ |    -
                     \     exp(1) /

          /        1   \
   2 exp(1) | x + ------ |
          \     exp(1) /    / /        1   \3/2 \
   ---------------------- + O| | x + ------ |    |
             3              \ \     exp(1) /    /

       1/2       1/2 /        1   \1/2
- 1 - 2    exp(1)    | x + ------ |    -
                     \     exp(1) /

          /        1   \
   2 exp(1) | x + ------ |
          \     exp(1) /    / /        1   \3/2 \
   ---------------------- + O| | x + ------ |    |
             3              \ \     exp(1) /    /
```

Background:

- Reference: R.M. Corless, D.J. Jeffrey and D.E. Knuth: "A sequence of Series for the Lambert W Function", in: Proceedings of ISSAC'97, Maui, Hawaii. W.W. Kuechlin (ed.). New York: ACM, pp. 197-204, 1997.

Changes:

- Series expansions can now be computed via `series`.

`last` – access a previously computed object

`%` returns the result of the last command.

`last(n)` or `%n` returns the result of the `n`th previous command.

Call(s):

- `last(n)`

- `%`
- `%n`

Parameters:

`n` — a positive integer

Return Value: a MuPAD object.

Further Documentation: Chapter 12 of the MuPAD Tutorial.

Related Functions: `HISTORY`, `history`

Details:

- By default, MuPAD stores the last 20 commands and their results in an internal history table. `last(n)` returns the result entry of the `n`th element in this table, counted from the end of the table. Thus `last(1)` returns the result of the last command, `last(2)` returns the result of the next to last one, etc. Instead of `last(n)` one can also write more briefly `%n`. Instead of `last(1)` or `%1`, one can use even more briefly `%`.
- The environment variable `HISTORY` determines the number of previous results that can be accessed at interactive level, i.e., the number of entries in the history table. In procedures, the length of this table is always 3, independent of the value of `HISTORY`. Thus admissible values for `n` are the integers between `1` and `HISTORY` at interactive level, and the integers `1, 2, 3` inside a procedure.

  Use `history` to access entries of the history table at interactive level directly, including the command that produced the corresponding result.
- The result returned by `last` or `%` is not evaluated again. Use the function `eval` to force a subsequent evaluation. See example 4.
- `last` behaves differently at interactive level and in procedures. At interactive level, compound statements, such as `for`, `repeat`, and `while` loops and `if` and `case` branching instructions, are stored in the history table as a whole. In procedures, the statements within a compound statement are stored in a separate history table of this procedure, but not the compound statement itself. See example 5.

  NOTE
- Commands and their results are stored in the history table even if the output is suppressed by a colon. Thus the result of `last(n)` may differ from the `n`th previous output that is visible on the screen at interactive level. See example 1.

- Commands appearing on the same input line lead to separate entries in the history table if they are separated by a colon or a semicolon. In contrast, an expression sequence is regarded as a single command. See example 2.
- Commands that are read from a file via `fread` or `read` are stored in the history table *before* the `fread` or `read` command itself. If the option *Plain* is used, then a separate history table is valid within the file, and the commands from the file do not appear in the history table of the enclosing context. See the help page of `history` for examples.
- Using `last` in procedures is generally considered bad programming style and is therefore deprecated. Future MuPAD releases may no longer support the use of `last` within procedures.
- If the abbreviated syntax `%n` is used, then `n` must be a positive integer literally. If this is not the case, but `n` evaluates to a positive integer, use the equivalent functional notation `last(n)` (see example 3).
- `last` is a function of the system kernel.

Example 1. Here are some examples for using `last` at interactive level. Note that `last(n)` refers to the `n`th previously computed result, whether it was displayed or not:

```
>> a := 42;
   last(1), %, %1

                        42

                    42, 42, 42

>> a := 34: b := 56: last(2) = %2

                      34 = 34
```

Example 2. Commands appearing on one input line lead to separate entries in the history table:

```
>> "First command"; 11: 22; 33:

                  "First command"

                        22

>> last(1), last(2);

                      33, 22
```

If a sequence of commands is bracketed, it is regarded as a single command:

```
>> "First command"; (11: 22; 33:)

                          "First command"

                                33

>> last(1), last(2);

                       33, "First command"
```

An expression sequence is also regarded as a single command:

```
>> "First command"; 11, 22, 33;

                          "First command"

                            11, 22, 33

>> last(1), last(2);

                   11, 22, 33, "First command"
```

Example 3. Due of the fact that the MuPAD parser expects a number after the `%` sign, there is a difference between the use of `%` and `last`. `last` can be called with an expression that evaluates to a positive integer:

```
>> n := 2: a := 35: b := 56: last(n)

                                35
```

If you try the same with `%`, an error occurs:

```
>> n := 2: a := 35: b := 56: %n

 Error: Unexpected 'identifier' [line 2, col 0]
```

Example 4. The result of `last` is not evaluated again:

```
>> delete a, b:
   c := a + b + a: a:= b: last(2)

                              2 a + b
```

Use `eval` to enforce the evaluation:

```
>> eval(%)

                                3 b
```

Example 5. We demonstrate the difference between the use of `last` at interactive level and in procedures:

```
>> 1: for i from 1 to 3 do i: print(last(1)): end_for:

                                   1

                                   1

                                   1
```

Here `last(1)` refers to the most recent entry in the history table, which is the `1` executed before the `for` loop. We can also verify this by inspecting the history table after these commands. The command `history` returns a list with two elements. The first entry is a previously entered MuPAD command, and the second entry is the result of this command returned by MuPAD. You see that the history table contains the whole `for` loop as a single command:

```
>> history(history() - 1), history(history())

                 [1, 1], [(for i from 1 to 3 do
                   i;
                   print(last(1))
                 end_for), null()]
```

However, if the `for` loop defined above is executed inside a procedure, then we obtain a different result. In the following example, `last(1)` refers to the last evaluated expression, namely the `i` inside the loop:

```
>> f := proc()
   begin
     1: for i from 1 to 3 do i: print(last(1)): end_for
   end_proc:

>> f():

                                   1

                                   2

                                   3
```

The command `history` refers only to the interactive inputs and their results:

```
>> history(history())

                              [f(), null()]
```

`lasterror` – reproduce the last error

`lasterror()` reproduces the last error that occurred in the current MuPAD session.

Call(s):

- `lasterror()`

Related Functions: `error`, `traperror`

Details:

- Typically, `lasterror` is used to reproduce errors that were caught by `traperror`. Cf. example 2.
- `lasterror` is a function of the system kernel.

Example 1. We produce an error:

```
>> x := 0: y := 1/x

 Error: Division by zero
```

This error may be reproduced by `lasterror`:

```
>> lasterror()

 Error: Division by zero
```

A further error is produced:

```
>> error("my error")

 Error: my error

>> lasterror()

 Error: my error

>> delete x, y:
```

Example 2. The following procedure `mysin` computes the sine function of its argument. In case of an error produced by the system function `sin`, it prints information on the argument and reproduces the error:

```
>> mysin := proc(x)
   local result;
   begin
     if traperror((result := sin(x))) = 0 then
        return(result)
     else
        print(Unquoted, "the following error occurred " .
                        "when calling sin(".expr2text(x)."):");
        lasterror()
     end_if:
   end:
```

Indeed, the system's sine function produces an error for large floating point arguments:

```
>> mysin(1.0*10^100)

     the following error occurred when calling sin(1.0e100):
 Error: Loss of precision;
 during evaluation of 'sin'

>> delete mysin:
```

`lcm` – the least common multiple of polynomials

`lcm(p, q, ...)` returns the least common multiple of the polynomials $p, q, \ldots$

Call(s):

- `lcm(p, q, ...)`
- `lcm(f, g, ...)`

Parameters:

`p, q, ...` — polynomials of type `DOM_POLY`
`f, g, ...` — polynomial expressions

Return Value: a polynomial, a polynomial expression, or the value `FAIL`.

Overloadable by: `p, q, f, g`

Related Functions: `content`, `factor`, `gcd`, `gcdex`, `icontent`, `ifactor`, `igcd`, `igcdex`, `ilcm`, `poly`

Details:

- `lcm(p, q, ...)` calculates the greatest common divisor of any number of polynomials. The coefficient ring of the polynomials may either be the integers or the rational numbers, ***Expr***, a residue class ring ***IntMod***`(n)` with a prime number `n`, or a domain.

  All polynomials must have the same indeterminates and the same coefficient ring.

- Polynomial expressions are converted to polynomials. See `poly` for details. `FAIL` is returned if an argument cannot be converted to a polynomial.

- The return value is of the same type as the input polynomials, i.e., either a polynomial of type `DOM_POLY` or a polynomial expression.

- `lcm` returns 1 if all arguments are 1 or -1, or if no argument is given. If at least one of the arguments is 0, then `lcm` returns 0.

- Use `ilcm` if all arguments are known to be integers, since it is much faster than `lcm`.

Example 1. The least common multiple of two polynomial expressions can be computed as follows:

```
>> lcm(x^3 - y^3, x^2 - y^2);

                      4    4       3    3
                     y  - x  + x y  - x  y
```

One may also choose polynomials as arguments:

```
>> p := poly(x^2 - y^2, [x, y], IntMod(17)):
   q := poly(x^2 - 2*x*y + y^2, [x, y], IntMod(17)):
   lcm(p, q)

               3    2         2    3
         poly(x  - x  y - x y  + y , [x, y], IntMod(17))

>> delete f, g, p, q:
```

`lcoeff` – the leading coefficient of a polynomial

`lcoeff(p)` returns the leading coefficient of the polynomial `p`.

Call(s):

- `lcoeff(p <, vars> <, order>)`

Parameters:

`p`	—	a polynomial of type `DOM_POLY` or a polynomial expression
`vars`	—	a list of indeterminates of the polynomial: typically, identifiers or indexed identifiers
`order`	—	the term ordering: either *LexOrder*, or *DegreeOrder*, or *DegInvLexOrder*, or a user-defined term ordering of type `Dom::MonomOrdering`. The default is the lexicographical ordering *LexOrder*.

Return Value: an element of the coefficient domain of the polynomial or `FAIL`.

Overloadable by: `p`

Related Functions: `coeff`, `collect`, `degree`, `degreevec`, `ground`, `ldegree`, `lmonomial`, `lterm`, `nterms`, `nthcoeff`, `nthmonomial`, `nthterm`, `poly`, `poly2list`, `tcoeff`

Details:

- The argument `p` can either be a polynomial expression, or a polynomial generated by `poly`, or an element of some polynomial domain overloading `lcoeff`.
- If a list of indeterminates is provided, then `p` is regarded as a polynomial in these indeterminates. Note that the specified list does not have to coincide with the indeterminates of the input polynomial. Cf. example 1.
- The returned coefficient is "leading" with respect to the lexicographical ordering, unless a different ordering is specified via the argument `order`. Cf. example 2.
- The result of `lcoeff` is not fully evaluated. Evaluation can be enforced by the function `eval`. Cf. example 3.
- `lcoeff` returns `FAIL` if the input polynomial cannot be converted to a polynomial in the specified indeterminates. Cf. example 4.
- With the orderings *LexOrder*, *DegreeOrder* and *DegInvLexOrder*, `lcoeff` calls a fast kernel function. Other orderings are handled by slower library functions.

Example 1. We demonstrate how the indeterminates influence the result:

```
>> p := 2*x^2*y + 3*x*y^2 + 6:
   lcoeff(p), lcoeff(p, [x, y]), lcoeff(p, [y, x])

                              3, 2, 3
```

Note that the indeterminates passed to `lcoeff` will be used, even if the polynomial provides different indeterminates :

```
>> p := poly(2*x^2*y + 3*x*y^2, [x, y]):
   lcoeff(p), lcoeff(p, [x, y]), lcoeff(p, [y, x]),
   lcoeff(p, [y]), lcoeff(p, [z])

                                   2          2
                  2, 2, 3, 3 x, 2 x  y + 3 x y

>> delete p:
```

Example 2. We demonstrate how various orderings influence the result:

```
>> p := poly(5*x^4 + 4*x^3*y*z^2 + 3*x^2*y^3*z + 2, [x, y, z]):
   lcoeff(p), lcoeff(p, DegreeOrder), lcoeff(p, DegInvLexOrder)

                              5, 4, 3
```

The following call uses the reverse lexicographical order on 3 indeterminates:

```
>> lcoeff(p, Dom::MonomOrdering(RevLex(3)))

                                 3

>> delete p:
```

Example 3. The result of `lcoeff` is not fully evaluated:

```
>> p := poly(a*x^2 + 27*x, [x]): a := 5:
   lcoeff(p, [x]), eval(lcoeff(p, [x]))

                               a, 5

>> delete p, a:
```

Example 4. We define a polynomial over the integers modulo 7:

```
>> p := poly(3*x, [x], Dom::IntegerMod(7)): lcoeff(p)
```

```
3 mod 7
```

This polynomial cannot be regarded as a polynomial with respect to another indeterminate, because the "coefficient" `3*x` cannot be interpreted as an element of the coefficient ring `Dom::IntegerMod(7)`:

```
>> lcoeff(p, [y])
```

```
FAIL
```

```
>> delete p:
```

`ldegree` – the lowest degree of the terms in a polynomial

`ldegree(p)` returns the lowest total degree of the terms of the polynomial `p`.

`ldegree(p, x)` returns the lowest degree of the terms in `p` with respect to the variable `x`.

Call(s):

- `ldegree(p)`
- `ldegree(p, x)`
- `ldegree(f <, vars>)`
- `ldegree(f <, vars>, x)`

Parameters:

`p`	—	a polynomial of type `DOM_POLY`
`f`	—	a polynomial expression
`vars`	—	a list of indeterminates of the polynomial: typically, identifiers or indexed identifiers
`x`	—	an indeterminate

Return Value: a nonnegative number. `FAIL` is returned if the input cannot be converted to a polynomial.

Overloadable by: `p`, `f`

Related Functions: `coeff`, `degree`, `degreevec`, `ground`, `lcoeff`, `lmonomial`, `lterm`, `nterms`, `nthcoeff`, `nthmonomial`, `nthterm`, `poly`, `poly2list`, `tcoeff`

Details:

- If the first argument `f` is not element of a polynomial domain, then `ldegree` converts the expression to a polynomial via `poly(f)`. If a list of indeterminates is specified, then the polynomial `poly(f, vars)` is considered.
- `ldegree(f, vars, x)` returns 0 if `x` is not an element of `vars`.
- The low degree of the zero polynomial is defined as 0.
- `ldegree` is a function of the system kernel.

Example 1. The lowest total degree of the terms in the following polynomial is computed:

```
>> ldegree(x^3 + x^2*y^2)

                                   3
```

The next call regards the expression as a polynomial in `x` with a parameter `y`:

```
>> ldegree(x^3 + x^2*y^2, x)

                                   2
```

The next expression is regarded as a bi-variate polynomial in `x` and `z` with coefficients containing the parameter `y`. The total degree with respect to `x` and `z` is computed:

```
>> ldegree(x^3*z^2 + x^2*y^2*z, [x, z])

                                   3
```

We compute the low degree with respect to `x`:

```
>> ldegree(x^3*z^2 + x^2*y^2*z, [x, z], x)

                                   2
```

A polynomial in `x` and `z` is regarded constant with respect to any other variable, i.e., its corresponding degree is 0:

```
>> ldegree(poly(x^3*z^2 + x^2*y^2*z, [x, z]), y)

                                   0
```

`length` – the "length" of a MuPAD object (heuristic complexity)

`length(object)` returns an integer indicating the complexity of the object.

Call(s):

- length(object)

Parameters:

`object` — an arbitrary MuPAD object

Return Value: a nonnegative integer.

Related Functions: `nops`, `op`

Details:

- The (heuristic) complexity of an object may be useful in algorithms that need to predict the complexity and time for manipulating objects. E.g., a symbolic Gaussian algorithm for solving linear equations prefers Pivot elements of small complexity.

- The length of an object is determined as follows:

  - Objects of domain type `DOM_BOOL`, `DOM_DOMAIN`, `DOM_EXEC`, `DOM_FAIL`, `DOM_FLOAT`, `DOM_FUNC_ENV`, `DOM_IDENT`, `DOM_NIL`, `DOM_VAR`, and `DOM_PROC_ENV` are regarded as "atomic". They have length 1. In particular, the length of identifiers and real floating point numbers is 1.
  - The length of an integer is the number of decimal digits.
  - The length of a string is the number of its characters.
  - The length of composite objects such as complex numbers, rational numbers, arithmetical expressions, lists, sets, arrays, tables etc. is the sum of the lengths of the operands plus 1.

- `length()` yields 0.

- `length` does *not* return the number of elements or entries in sets, lists or tables. Use `nops` instead! NOTE

- `length` is a function of the system kernel.

Example 1. Intuitively, the length measures the complexity of an object:

```
>> length(1 + x) < length(x^3 + exp(a - b)/ln(45 - t) - 1234*I)

                            3 < 25
```

Example 2. We compute the lengths of some simple objects:

```
>> length(1.2), length(-1234.5), length(123456), length(-123456)

                          1, 1, 6, 6

>> length(17), length(123), length(17/123)

                            2, 3, 6

>> length(12), length(123), length(12 + 123*I)

                            2, 3, 6

>> length(x), length(x^2), length(x^12345)

                            1, 3, 7

>> length("123"), length("")

                             3, 0

>> length(x), length(a_long_name)

                             1, 1
```

Example 3. The length of an array is the sum of the lengths of all its elements plus 1:

```
>> A := array(1..2, [x, y]): length(A) = length(x) + length(y) + 1

                             3 = 3

>> A[1] := 12345: length(A) = length(12345) + length(y) + 1

                             7 = 7

>> delete A:
```

Example 4. The operands of a table are the equations associating indices and entries. The length of each operand is the length of the index plus the length of the corresponding entry plus 1:

```
>> T[1] := 45: T

                            table(
                              1 = 45
                            )
```

```
>> length(T) = length(1 = 45) + 1

                                5 = 5

>> delete T:
```

`level` – evaluate an object with a specified substitution depth

`level(object, n)` evaluates `object` with substitution depth `n`.

Call(s):

- `level(object)`
- `level(object, n)`

Parameters:

`object` — any MuPAD object
`n` — a nonnegative integer less than 2^{31}

Return Value: the evaluated object.

Further Documentation: Chapter 5 of the MuPAD Tutorial.

Related Functions: `context`, `eval`, `hold`, `indexval`, `LEVEL`, `MAXLEVEL`, `val`

Details:

- When a MuPAD object is evaluated, identifiers occurring in it are replaced by their values. This happens recursively, i.e., if the values themselves contain identifiers, then these are replaced as well. `level` serves to evaluate an object with a specified recursion depth for this substitution process.
- With `level(object, 0)`, `object` is evaluated without replacing any identifier occurring in it by its value. In most cases, but not always, this equivalent to `hold(object)`, and `object` is returned unevaluated. See example 3.
- With `level(object, 1)`, all identifiers occurring in `object` are replaced by their values, but not recursively, and then all function calls in the result of the substitution are executed. This is how objects are evaluated within a procedure by default.

- The call `level(object)` is equivalent to `level(object, MAXLEVEL)`, i.e., identifiers occurring in `object` are recursively replaced by their values up to substitution depth `MAXLEVEL - 1`, and an error occurs if the substitution depth `MAXLEVEL` is reached. Usually, this leads to a complete evaluation of `object`. See example 1.

- You can use `level` without a second argument to request the complete evaluation of an object not containing local variables or formal parameters within a procedure. This may be necessary since by default, objects are evaluated with substitution depth `1` within procedures. See example 2.

  Otherwise, it should never be necessary to use `level`.

- `level` does not affect the evaluation of local variables and formal parameters, of type `DOM_VAR`, in procedures. When such a local variable occurs in `object`, then it is always replaced by its value, independent of the value of `n`, and the value is not further recursively evaluated. See example 2.

- `level` works by temporarily setting the value of `LEVEL` to `n`, or to $2^{31} - 1$ if `n` is not given. However, the value of `MAXLEVEL` remains unchanged. If the substitution depth `MAXLEVEL` is reached, then an error message is returned. See `LEVEL` and `MAXLEVEL` for more information on these environment variables.

- In contrast to most other functions, `level` does not flatten its first argument if it is an expression sequence. See example 5.

- `level` does not recursively descend into `arrays`, `tables`, `matrices` or `polynomials`. Use the call `map(object, eval)` to evaluate the entries of an array, a table, a matrix or `mapcoeffs(object, eval)` to evaluate the coefficients of a polynomial. Cf. example 4 and example 6.

  Further information concerning the evaluation of arrays, tables, matrices or polynomials can be found on the `eval` help page.

- The maximal substitution depth of `level` depends on the environment variable `MAXLEVEL`, while the maximum evaluation depth of the function `eval` depends on the environment variable `LEVEL`. See example 7.

- Because `eval` evaluates the result again there is a difference between evaluating an expression with depth n by `level` in comparison with `eval`. See example 7.

- As mentioned `level` does not affect the evaluation of local variables and formal parameters, of type `DOM_VAR`, in procedures. Here `eval` behaves different. See example 7 and the `eval` help page for more information.

- The result of `level(hold(x))` is always `x`, because a full evaluation of `hold(x)` leads to `x`. The same does not hold for `eval(hold(x))`, because `eval` first evaluates its argument and then evaluates the result again.

- The evaluation of elements of a user-defined `domain` depends on the implementation of the domain. Usually domain elements remain unevaluated by `level`. If the domain has a slot `"evaluate"`, the corresponding slot routine is called with the domain element as argument at each evaluation, and hence it is called once when `level` is invoked. Cf. example 8.

- `level` is a function of the system kernel.

Example 1. We demonstrate the effect of `level` for various values of the second parameter:

```
>> delete a0, a1, a2, a3, a4, b: b := b + 1:
   a0 := a1: a1 := a2 + 2: a2 := a3 + a4: a3 := a4^2: a4 := 5:

>> hold(a0), hold(a0 + a2), hold(b);
   level(a0, 0), level(a0 + a2, 0), level(b, 0);
   level(a0, 1), level(a0 + a2, 1), level(b, 1);
   level(a0, 2), level(a0 + a2, 2), level(b, 2);
   level(a0, 3), level(a0 + a2, 3), level(b, 3);
   level(a0, 4), level(a0 + a2, 4), level(b, 4);
   level(a0, 5), level(a0 + a2, 5), level(b, 5);
   level(a0, 6), level(a0 + a2, 6), level(b, 6);

                        a0, a0 + a2, b

                        a0, a0 + a2, b

                    a1, a1 + a3 + a4, b + 1

                                 2
                  a2 + 2, a2 + a4  + 7, b + 2

               a3 + a4 + 2, a3 + a4 + 32, b + 3

                     2        2
                   a4  + 7, a4  + 37, b + 4

                         32, 62, b + 5

                         32, 62, b + 6
```

Evaluating `object` by just typing `object` at the command prompt is equivalent to `level(object, LEVEL)`:

```
>> LEVEL := 2: MAXLEVEL := 4: a0, a2, b;
   level(a0, LEVEL), level(a2, LEVEL), level(b, LEVEL)
```

```
                    2
a2 + 2, a4  + 5, b + 2

                    2
a2 + 2, a4  + 5, b + 2
```

If the second argument is omitted, then this corresponds to a complete evaluation up to substitution depth `MAXLEVEL - 1`:

```
>> level(a0)

 Error: Recursive definition [See ?MAXLEVEL]

>> level(a2)

                             30

>> level(b)

 Error: Recursive definition [See ?MAXLEVEL]

>> delete LEVEL, MAXLEVEL:
```

Example 2. We demonstrate the behavior of `level` in procedures:

```
>> delete a, b, c: a := b: b := c: c := 42:
   p := proc()
     local x;
   begin
     x := a:
     print(level(x, 0), x, level(x, 2), level(x)):
     print(level(a, 0), a, level(a, 2), level(a)):
   end_proc:
   p()

                         b, b, b, b

                         a, b, c, 42
```

Since `a` is evaluated with the default substitution depth `1`, the assignment `x:=a` sets the value of the local variable `x` to the unevaluated identifier `b`. You can see that any evaluation of `x`, whether `level` is used or not, simply replaces `x` by its value `b`, but no further recursive evaluation happens. In contrast, evaluation of the identifier `a` takes place with the default substitution depth `1`, and `level(a, 2)` evaluates it with substitution depth `2`.

Thus `level` without a second argument can be used to request the complete evaluation of an object not containing any local variables or formal parameters.

Example 3. There are some rare cases where `level(object, 0)` and `hold(object)` behaves different. This is the case if `object` is not an identifier, e.g., a nameless function, because `level` influences only the evaluation of identifiers:

```
>> level((x -> x^2)(2),0), hold((x -> x^2)(2))

                        4, (x -> x^2)(2)
```

For the same reason `level(object, 0)` and `hold(object)` behave differently if `object` is a local variable of a procedure:

```
>> f:=proc() local x; begin
     x := 42;
     hold(x), level(x, 0);
   end_proc:
   f();
   delete f:

                        DOM_VAR(0,2), 42
```

Example 4. In contrast to lists and sets, evaluation of an array does not evaluate its entries. Thus `level` has no effect for arrays either. The same holds for tables and matrices. Use `map` to evaluate all entries of an array. On the `eval` help page further examples can be found:

```
>> delete a, b:
   L := [a, b]:  A := array(1..2, L):  a := 1:   b := 2:
   L, A, level(A), map(A, level), map(A, eval)

                +-     -+  +-     -+  +-     -+  +-     -+
        [1, 2], | a, b |, | a, b |, | a, b |, | 1, 2 |
                +-     -+  +-     -+  +-     -+  +-     -+
```

Example 5. The first argument of `level` may be an expression sequence, which is not flattened. However, it must be enclosed in parentheses:

```
>> delete a, b: a := b: b := 3:
   level((a, b), 1);
   level(a, b, 1)

                              b, 3
 Error: Wrong number of arguments [level]
```

Example 6. `Polynomials` are inert when evaluated, and so `level` has no effect:

```
>> delete a, x: p := poly(a*x, [x]): a := 2: x := 3:
   p, level(p)

                  poly(a x, [x]), poly(a x, [x])
```

Use `mapcoeffs` and the function `eval` to evaluate all coefficients:

```
>> mapcoeffs(p, eval)

                          poly(2 x, [x])
```

If you want to substitute a value for the indeterminate `x`, use `evalp`:

```
>> delete x: evalp(p, x = 3)

                                3 a
```

As you can see, the result of an `evalp` call may contain unevaluated identifiers, and you can evaluate them by an application of `eval`. It is necessary to use `eval` instead of `level` because `level` does not evaluate its result:

```
>> eval(evalp(p, x = 3))

                                 6
```

Example 7. The subtle difference between `level` and `eval` is shown. The evaluation depth of `eval` is limited by the environment variable `LEVEL`. `level` pays no attention to `LEVEL`, but rather continues evaluating its argument either as many times as the second argument implies or until it has been evaluated completely:

```
>> delete a0, a1, a2, a3:
   a0 := a1 + a2: a1 := a2 + a3: a2 := a3^2 - 1: a3 := 5:
   LEVEL := 1:
   eval(a0), level(a0);

                                  2
                    a2 + a3 + a3  - 1, 53
```

If the evaluation depth exceeds the value of `MAXLEVEL`, an error is raised in both cases:

```
>> delete LEVEL:
   MAXLEVEL := 3:
   level(a0);
```

```
 Error: Recursive definition [See ?MAXLEVEL]
```

```
>> delete LEVEL:
   MAXLEVEL := 3:
   eval(a0);
   delete MAXLEVEL:
```

```
 Error: Recursive definition [See ?MAXLEVEL]
```

It is not the same evaluating an expression `ex` with `eval` and an evaluation depth n and by `level((ex, n))`, because `eval` evaluates its result:

```
>> LEVEL := 2: eval(a0), level(a0, 2);
   delete LEVEL:
```

```
                                     2
                53, a2 + a3 + a3  - 1
```

`level` does not affect the evaluation of local variables of type `DOM_VAR` while `eval` evaluates them with evaluation depth `LEVEL`, which is one in a procedure:

```
>> p := proc()
     local x;
   begin
     x := a0:
     print(eval(x), level(x)):
   end_proc:
   p()
```

```
                      2
          a2 + a3 + a3  - 1, a1 + a2
```

Example 8. The evaluation of an element of a user-defined domain depends on the implementation of the domain. Usually it is not further evaluated:

```
>> delete a: T := newDomain("T"):
   e := new(T, a): a := 1:
   e, level(e), map(e, level), val(e)
```

```
          new(T, a), new(T, a), new(T, a), new(T, a)
```

If the slot `"evaluate"` exists, the corresponding slot routine is called for a domain element each time it is evaluated. We implement the routine `T::evaluate`, which simply evaluates all internal operands of its argument, for our domain `T`. The unevaluated domain element can still be accessed via `val`:

```
>> T::evaluate := x -> new(T, eval(extop(x))):
   e, level(e), map(e, level), val(e);
```

```
            new(T, 1), new(T, 1), new(T, 1), new(T, a)
```

```
>> delete e, T:
```

---

lhs, rhs – the left, respectively right hand side of equations, inequalities, relations, intervals, and ranges

`lhs(f)` returns the left hand side of `f`.

`rhs(f)` returns the right hand side of `f`.

Call(s):

- `lhs(f)`
- `rhs(f)`

Parameters:

`f` — an equation `x = y`, an inequality `x <> y`, a relation `x < y`, a relation `x <= y`, an interval `x...y`, or a range `x..y`

Return Value: an arithmetical expression.

Overloadable by: `f`

Related Functions: `op`

Details:

- The calls `lhs(f)` and `rhs(f)` are equivalent to the direct calls `op(f, 1)` and `op(f, 2)`, respectively, of the operand function `op`.

Example 1. We extract the left and right hand sides of various objects:

```
>> lhs(x = sin(2)), lhs(3.14 <> PI), lhs(x + 3 < 2*y),
   rhs(a <= b), rhs(m-1..n+1)

                    x, 3.14, x + 3, b, n + 1
```

The operands of an expression depend on its internal representation. In particular, a "greater" relation is always converted to the corresponding "less" relation:

```
>> y > -infinity; lhs(y > -infinity)
```

```
                    -infinity < y

                      -infinity

>> y >= 4; rhs(y >= 4)

                         4 <= y

                            y
```

Example 2. We extract the left and right hand sides of the solution of the following system:

```
>> s := solve({x + y = 1, 2*x - 3*y = 2})

                   {[x = 1, y = 0]}

>> map(op(s), lhs) =  map(op(s), rhs)

                    [x, y] = [1, 0]
```

Calls to `lhs` and `rhs` may be easier to read than the equivalent calls to the operand function `op`:

```
>> map(op(s), op, 1) = map(op(s), op, 2)

                    [x, y] = [1, 0]
```

However, direct calls to `op` should be preferred inside procedures for higher efficiency.

```
>> delete s:
```

`limit` – compute a limit

`limit(f, x = x0<, Real>)` computes the bidirectional limit $\lim\limits_{\substack{x\to x_0\\ x-x_0\in\mathbb{R}\setminus\{0\}}} f(x)$.

`limit(f, x = x0, Left)` computes the one-sided limit $\lim\limits_{\substack{x\to x_0\\ x<x_0}} f(x)$.

`limit(f, x = x0, Right)` computes the one-sided limit $\lim\limits_{\substack{x\to x_0\\ x>x_0}} f(x)$.

Call(s):

- `limit(f, x <= x0> <, dir>)`

Parameters:

- `f` — an arithmetical expression representing a function in `x`
- `x` — an identifier
- `x0` — the limit point: an arithmetical expression, possibly `infinity` or `-infinity`

Options:

- `dir` — either *`Left`*, *`Right`*, or *`Real`*. This controls the direction of the limit computation. The option *`Real`* is the default case and means the bidirectional limit (i.e., there is no need to specify this option).

Return Value: an arithmetical expression, an interval of type `Dom::Interval`, an expression of type `"limit"`, or `FAIL`.

Side Effects: The function is sensitive to the environment variable `ORDER`, which determines the default number of terms in series computations (see `series` and example 6 below).

Properties of identifiers set by `assume` are taken into account.

Overloadable by: `f`

Related Functions: `asympt`, `diff`, `discont`, `int`, `O`, `series`, `taylor`

Details:

- `limit(f, x = x0`<`, `*`Real`*>`)` computes the bidirectional limit of `f` when `x` tends to `x0` on the real axis. The limit point `x0` may be omitted, in which case `limit` assumes `x0 = 0`.

  If the limit point `x0` is ∞ or $-\infty$, then the limit is taken from the left to ∞ or from the right to $-\infty$, respectively.

  If the left and right limits are different, then `undefined` is returned; see example 2.

- `limit(f, x = x0, Left)` returns the limit when `x` tends to `x0` from the left. `limit(f, x = x0, Right)` returns the limit when `x` tends to `x0` from the right. See example 2.

- If the system cannot compute a limit, but can assert that the function `f` is bounded when `x` approaches `x0`, then a bounding interval, of type `Dom::Interval`, for `f(x)` in a sufficiently small neighborhood of `x0` is returned. This may happen, e.g., if `f` oscillates infinitesimally fast in the neighborhood of `x0`. Note, however, that the boundaries need not be equal to the limes inferior and the limes superior of `f` for $x \to x_0$. See example 4.

- If the limit cannot be computed, then the system returns a symbolic `limit` call (see example 3).
- If `f` contains parameters, then `limit` reacts to properties of those parameters set by `assume`; see example 5. If the limit cannot be computed without additional assumptions about the parameters, then `limit` indicates this by a warning.
- Internally, `limit` tries to determine the limit from a series expansion of `f` around `x = x0` computed via `series`. If the number of terms in the series expansion is too small to compute the limit, then `limit` returns `FAIL`. In such a case, it may be necessary to increase the value of the environment variable `ORDER` in order to find the limit (see example 6).
- `limit` works on a symbolic level and should not be called with arguments containing floating point arguments. NOTE

Example 1. The following command computes $\lim_{x \to 0} \frac{1 - \cos x}{x^2}$:

```
>> limit((1 - cos(x))/x^2, x)

                              1/2
```

A possible definition of e is given by the limit of the sequence $\left(1 + \frac{1}{n}\right)^n$ for $n \to \infty$:

```
>> limit((1 + 1/n)^n, n = infinity)

                             exp(1)
```

Here is a more complex example:

```
>> limit(
     (exp(x*exp(-x)/(exp(-x) + exp(-2*x^2/(x+1)))) - exp(x))/x,
     x = infinity
   )

                             -exp(2)
```

Example 2. The bidirectional limit of $f(x) = 1/x$ for $x \to 0$ does not exist:

```
>> limit(1/x, x = 0)

                            undefined
```

You can compute the one-sided limits from the left and from the right by passing the options *`Left`* and *`Right`*, respectively:

```
>> limit(1/x, x = 0, Left),
   limit(1/x, x = 0, Right)

                              -infinity, infinity
```

Example 3. If `limit` is not able to compute the limit, then a symbolic `limit` call is returned:

```
>> delete f: limit(f(x), x = infinity)

                          limit(f(x), x = infinity)
```

Example 4. The function $\sin(x)$ oscillates for $x \to \infty$. The limes inferior and the limes superior are -1 and 1, respectively:

```
>> limit(sin(x), x = infinity)

                                   [-1, 1]
```

In fact, for $x \to \infty$ the function `f=sin(x)` assumes every value in the returned interval infinitely often. This need not be the case in general.

The boundaries of the interval returned by `limit` need not coincide with the limit superior and the limit inferior, respectively. In the following example, the limit inferior and the limit superior are in fact $-\sqrt{2}$ and $\sqrt{2}$, respectively:

```
>> limit(sin(x) + cos(x), x = infinity)

                                   [-2, 2]
```

Example 5. `limit` is not able to compute the limit of x^n for $x \to \infty$ without additional information about the parameter n:

```
>> delete n: limit(x^n, x = infinity)

 Warning: cannot determine sign of n [stdlib::limit::limitIV::_\
 power]

                                  n
                           limit(x , x = infinity)
```

However, for $n > 0$ the limit exists and equals ∞. We use `assume` to achieve this:

```
>> assume(n > 0): limit(x^n, x = infinity)
```

```
infinity
```

Similarly, the limit is zero for $n < 0$:

```
>> assume(n < 0): limit(x^n, x = infinity)

0
```

Example 6. It may be necessary to increase the value of the environment variable `ORDER` in order to find the limit, as in the following example:

```
>> limit((sin(tan(x^2)) - tan(sin(x^2)))/x^14, x = 0)

 Warning: ORDER seems to be not big enough for series \
 computation [stdlib::limit::lterm]

FAIL
```

```
>> ORDER := 8:
   limit((sin(tan(x^2)) - tan(sin(x^2)))/x^14, x);
   delete ORDER:

-1/30
```

Background:

- If a limit cannot be computed, then `limit` issues a warning with a possible reason, as shown in examples 5 and 6. You may want to suppress these warnings when you call `limit` from within your own procedures. You can control this by means of the procedure `stdlib::limit::printWarnings`.

  The calls `stdlib::limit::printWarnings(TRUE)` and `stdlib::limit::printWarnings(FALSE)` switch the warnings that `limit` issues on and off, respectively, and return the previous setting. The command `stdlib::limit::printWarnings()` returns the current setting, which is `TRUE` by default.

- `limit` first tries a series computation to determine the limit. If this fails, then an algorithm based on the thesis of Dominik Gruntz: “On Computing Limits in a Symbolic Manipulation System”, Swiss Federal Institute of Technology, Zurich, Switzerland, 1995, is used.

Changes:

- One may specify the option *Real* to compute bidirectional limits.

`linsolve` – solve a system of linear equations

`linsolve(eqs, vars)` solves a system of linear equations with respect to the unknowns `vars`.

Call(s):

- `linsolve(eqs)`
- `linsolve(eqs, vars)`
- `linsolve(eqs, vars,` *ShowAssumptions*`)`
- `linsolve(eqs, vars,` *Domain* `= R)`

Parameters:

`eqs`	—	a list or a set of linear equations or arithmetical expressions
`vars`	—	a list or a set of unknowns to solve for: typically identifiers or indexed identifiers

Options:

ShowAssumptions	—	additionally return information about internal assumptions that `linsolve` has made on symbolic parameters in `eqs`
Domain`= R`	—	solve the system over the field `R`, which must be a domain of category `Cat::Field`.

Return Value: Without the option *ShowAssumptions*, a list of simplified equations is returned. It represents the general solution of the system `eqs`. `FAIL` is returned if the system is not solvable.

With *ShowAssumptions*, a list `[Solution, Constraints, Pivots]` is returned. `Solution` is a list of simplified equations representing the general solution of `eqs`. The lists `Constraints` and `Pivots` contain equations and inequalities involving symbolic parameters in `eqs`. Internally, these were assumed to hold true when solving the system.

Related Functions: `linalg::matlinsolve`, `numeric::linsolve`, `solve`

Details:

- `linsolve(eqs,` $<$`, vars` $<$`,` *ShowAssumptions*$>>$`)` solves the linear system `eqs` with respect to the unknowns `vars`. If no unknowns are specified, then `linsolve` solves for all indeterminates in `eqs`; the unknowns are determined internally by `indets(eqs,PolyExpr)`.

- `linsolve(eqs, vars, Domain = R)` solves the system over the domain `R`, which must be a field, i.e., a domain of category `Cat::Field`.

  NOTE: Note that the return format does not allow to return kernel elements if elements of the domain `R` cannot be multiplied with the symbolic unknowns that span the kernel. In such a case, `linsolve` issues a warning and returns only a special solution. The kernel can be computed via `linalg::matlinsolve` for any field `R`.

- Each element of `eqs` must be either an equation or an arithmetical expression `f`, which is considered to be equivalent to the equation `f = 0`.

- The unknowns in `vars` need not be identifiers or indexed identifiers; expressions such as `sin(x)`, `f(x)`, or `y^(1/3)` are allowed as well. More generally, any expression accepted as indeterminate by `poly` is a valid unknown.

- If the option *ShowAssumptions* is not given and the system is solvable, then the return value is a list of equations of the form `var = value`, where `var` is one of the unknowns in `vars` and `value` is an arithmetical expression that does not involve any of the unknowns on the left hand side of a returned equation. Note that if the solution manifold has dimension greater than zero, then some of the unknowns in `vars` will occur on the right hand side of some returned equations, representing the degrees of freedom. See example 3.

- If `vars` is a list, then the solved equations are returned in the the same order as the unknowns in `vars`.

- The function `linsolve` can only solve systems of linear equations. Use `solve` for nonlinear equations.

- `linsolve` is an interface function to the procedures `numeric::linsolve` and `linalg::matlinsolve`. For more details see the help pages `numeric::linsolve`, `linalg::matlinsolve` and the background section of this help page.

- The system `eqs` is checked for linearity. Since such a test may be expensive, it is recommended to use `numeric::linsolve` or `linalg::matlinsolve` directly in cases you be sure that the system is linear.

- NOTE: `linsolve` does *not* react to properties of identifiers set by `assume`.

Option <`ShowAssumptions`>:

- With this option, a list `[Solution, Constraints, Pivots]` is returned. `Solution` is a list of solved equations representing the complete solution manifold of `eqs`, as described above. The lists `Constraints` and `Pivots` contain equations and inequalities involving symbolic parameters in eqs. Internally, these were assumed to hold true when solving the system. [FAIL, [], []] is returned, if the system is not solvable. See numeric::linsolve for more details.

Example 1. Equations and variables may be entered as sets or lists:

```
>> linsolve({x + y = 1, 2*x + y = 3}, {x, y}),
   linsolve({x + y = 1, 2*x + y = 3}, [x, y]),
   linsolve([x + y = 1, 2*x + y = 3], {x, y}),
   linsolve([x + y = 1, 2*x + y = 3], [x, y])

 [x = 2, y = -1], [x = 2, y = -1], [x = 2, y = -1],

    [x = 2, y = -1]
```

Also expressions may be used as variables:

```
>> linsolve({cos(x) + sin(x) = 1, cos(x) - sin(x) = 0},
            {cos(x), sin(x)})

                    [cos(x) = 1/2, sin(x) = 1/2]
```

Furthermore, indexed identifiers are valid, too:

```
>> linsolve({2*a[1] + 3*a[2] = 5, 7*a[2] + 11*a[3] = 13,
             17*a[3] + 19*a[1] = 23}, {a[1], a[2], a[3]})

        [a[1] = 691/865, a[2] = 981/865, a[3] = 398/865]
```

Next, we demonstrate the use of option *Domain* and solve a system over the field $\mathbb{Z}_{23}$ with it:

```
>> linsolve([2*x + y = 1, -x - y = 0],
            Domain = Dom::IntegerMod(23))

                    [x = 1 mod 23, y = 22 mod 23]
```

The following system does not have a solution:

```
>> linsolve({x + y = 1, 2*x + 2*y = 3}, {x, y})

                                FAIL
```

Example 2. We demonstrate the dependence of the solution of a systems from involved parameters:

```
>> eqs := [x + a*y = b, x + A*y = b]:
>> linsolve(eqs, [x, y])
```

```
                    [x = b, y = 0]
```

Note that for $a = A$ this is not the general solution. Using the option *ShowAssumptions* it turns out, that the above result is the general solution subject to the assumption $a \neq A$:

```
>> linsolve(eqs, [x, y], ShowAssumptions)
```

```
          [[x = b, y = 0], [], [A - a <> 0]]
```

```
>> delete eqs:
```

Example 3. If the solution of the linear system is not unique, then some of the unknowns are used as "free parameters" spanning the solution space. In the following example the unknown `z` is such a parameter. It does not turn up on the left hand side of the solved equations:

```
>> eqs := [x + y = z, x + 2*y = 0, 2*x - z = -3*y, y + z = 0]:
>> vars := [w, x, y, z]:
>> linsolve(eqs, vars)
```

```
                    [x = 2 z, y = -z]
```

Background:

- If the option *Domain* is not present, the system is solved by calling `numeric::linsolve` with the option *Symbolic*.
- If the option *Domain* = `R` is given and `R` is one of the two domains `Dom::ExpressionField()` or `Dom::Float`, then `numeric::linsolve` is used to compute the solution of the system. This function uses a sparse representation of the equations.

  Otherwise, `eqs` is first converted into a matrix and then solved by `linalg::matlinsolve`. A possibly sparse structure of the input system is not taken into account.

`lllint` – compute an LLL-reduced basis of a lattice

`lllint(A)` applies the LLL algorithm to the columns of the (not necessary square) matrix `A` with integer entries.

Call(s):

- `lllint(A, All)`
- `lllint(A)`

Parameters:

`A` — a matrix, given as a list of row vectors, each row being a list of integers

Return Value:

- With option *All*, a list `[T, B]` is returned, such that `B = A*T` and the columns of `B` form an LLL-reduced basis of the lattice spanned by the columns of `A`. Both `T` and `B` are given as lists of row vectors.
- Without option *All*, `lllint` only returns the transformation matrix `T` as a list of row vectors.

Related Functions: `linalg::basis`, `linalg::factorLU`, `linalg::factorQR`, `linalg::gaussElim`, `linalg::hermiteForm`, `linalg::orthog`

Details:

- `lllint` applies the LLL algorithm to the columns of the matrix `A`. Mathematically, the input matrix can be an arbitrary matrix with integer entries, possibly non-square, and possibly without full column rank.

  The matrix is passed to `lllint` in form of a list of row vectors, where each row vector is again a list of integers. The number of entries in each row must be equal. The matrices returned by `lllint` have this form as well.

  The computations are done entirely with integers and are both accurate and quite fast.
- You can use `matrix` to obtain a nicer screen output of the matrices. See example 1.
- `lllint` is a function of the system kernel.

Example 1. We apply the LLL algorithm to a matrix with two rows and three columns:

```
>> A := [[1, 2, 3], [4, 5, 6]]:
   [T, B] := lllint(A, All)

         [[[-1, 4], [1, -3], [0, 0]], [[1, -2], [1, 1]]]
```

We use `matrix` to print `A,T`, and `B` in a nicer form and check that indeed `B = A*T`:

```
>> matrix(A), matrix(T), matrix(B)

                          +-         -+
     +-          -+  |  -1,  4  |  +-        -+
     |  1, 2, 3  |  |           |  |  1, -2  |
     |           |, |   1, -3  |, |          |
     |  4, 5, 6  |  |           |  |  1,  1  |
     +-          -+  |   0,  0  |  +-        -+
                          +-         -+
```

```
>> matrix(B) = matrix(A)*matrix(T)

                 +-        -+   +-        -+
                 |  1, -2  |   |  1, -2  |
                 |         | = |         |
                 |  1,  1  |   |  1,  1  |
                 +-        -+   +-        -+
```

The result is to be interpreted as follows: the two column vectors

$$\begin{pmatrix} 1 \\ 1 \end{pmatrix} \text{ and } \begin{pmatrix} -2 \\ 1 \end{pmatrix}$$

form an LLL-reduced basis of the integer lattice generated by the three column vectors

$$\begin{pmatrix} 1 \\ 4 \end{pmatrix}, \quad \begin{pmatrix} 2 \\ 5 \end{pmatrix}, \text{ and } \begin{pmatrix} 3 \\ 6 \end{pmatrix}.$$

Without the option *All*, `lllint` returns only the transformation matrix `T`:

```
>> matrix(lllint([[1, 2, 3], [4, 5, 6]]))

                      +-         -+
                      |  -1,  4  |
                      |           |
                      |   1, -3  |
                      |           |
                      |   0,  0  |
                      +-         -+
```

Background:

- References:

A. K. Lenstra, H. W. Lenstra Jr., and L. Lovasz, Factoring polynomials with rational coefficients. Math. Ann. 261, 1982, pp. 515–534.

Joachim von zur Gathen and Jürgen Gerhard, Modern Computer Algebra. Cambridge University Press, 1999, Chapter 16.

George L. Nemhauser and Laurence A. Wolsey, Integer and Combinatorial Optimization. New York, Wiley, 1988.

A. Schrijver, Theory of Linear and Integer Programming. New York, Wiley, 1986.

`lmonomial` – the leading monomial of a polynomial

`lmonomial(p)` returns the leading monomial of the polynomial `p`.

Call(s):

- `lmonomial(p` <, `vars`> <, `order`> <, *`Rem`*>`)`

Parameters:

`p`	—	a polynomial of type `DOM_POLY` or a polynomial expression
`vars`	—	a list of indeterminates of the polynomial: typically, identifiers or indexed identifiers
`order`	—	the term ordering: either *`LexOrder`* or *`DegreeOrder`* or *`DegInvLexOrder`* or a user-defined term ordering of type `Dom::MonomOrdering`. The default is the lexicographical ordering *`LexOrder`*.

Options:

`Rem`	—	makes `lmonomial` return a list with the leading monomial and the "reductum".

Return Value: a polynomial of the same type as `p`. An expression is returned if `p` is an expression. `FAIL` is returned if the input cannot be converted to a polynomial. With *`Rem`*, a list of two polynomials is returned.

Overloadable by: `p`

Related Functions: `coeff`, `degree`, `degreevec`, `ground`, `lcoeff`, `ldegree`, `lterm`, `nterms`, `nthcoeff`, `nthmonomial`, `nthterm`, `poly`, `poly2list`, `tcoeff`

Details:

- The argument `p` can either be a polynomial expression, or a polynomial generated by `poly`, or an element of some polynomial domain overloading `lmonomial`.

- If a list of indeterminates is provided, then `p` is regarded as a polynomial in these indeterminates. Note that the specified list does not have to coincide with the indeterminates of the input polynomial. Cf. example 1.
- The returned monomial is "leading" with respect to the lexicographical ordering, unless a different ordering is specified via the argument `order`. Cf. example 2.
- `lmonomial` returns `FAIL` if the input polynomial cannot be converted to a polynomial in the specified indeterminates. Cf. example 4.
- The result of `lmonomial` is not fully evaluated. It can be evaluated by the functions `mapcoeffs` and `eval`. Cf. example 5.
- The leading monomial of the zero polynomial is the zero polynomial.
- For the orderings *LexOrder*, *DegreeOrder* and *DegInvLexOrder*, the result is computed by a fast kernel function. Other orderings are handled by slower library functions.

Option <Rem>:

- With this option, a list with two polynomials is returned: the leading monomial and the reductum. The reductum of a polynomial `p` is `p - lmonomial(p)`.

Example 1. We demonstrate how the indeterminates influence the result:

```
>> p := 2*x^2*y + 3*x*y^2 + 6:
   lmonomial(p), lmonomial(p, [x, y]), lmonomial(p, [y, x])

                           2     2         2
                        3 x y , 2 x  y, 3 x y
```

Note that the indeterminates passed to `lmonomial` will be used, even if the polynomial provides different indeterminates :

```
>> p := poly(2*x^2*y + 3*x*y^2, [x, y]):
   lmonomial(p), lmonomial(p, [x, y]), lmonomial(p, [y, x]),
   lmonomial(p, [y]), lmonomial(p, [z])

         2                      2
 poly(2 x  y, [x, y]), poly(2 x  y, [x, y]),

           2                          2
    poly(3 y  x, [y, x]), poly((3 x) y , [y]),

           2          2
    poly(2 x  y + 3 x y , [z])
```

```
>> delete p:
```

Example 2. We demonstrate how various orderings influence the result:

```
>> p := poly(5*x^4 + 4*x^3*y*z^2 + 3*x^2*y^3*z + 2, [x, y, z]):
   lmonomial(p), lmonomial(p, DegreeOrder),
   lmonomial(p, DegInvLexOrder)

        4                      3    2
 poly(5 x , [x, y, z]), poly(4 x  y z , [x, y, z]),

           2  3
    poly(3 x  y  z, [x, y, z])
```

The following call uses the reverse lexicographical order on 3 indeterminates:

```
>> lmonomial(p, Dom::MonomOrdering(RevLex(3)))

                           2  3
                   poly(3 x  y  z, [x, y, z])

>> delete p:
```

Example 3. We compute the reductum of a polynomial:

```
>> p := poly(2*x^2*y + 3*x*y^2 + 6, [x, y]):
   q := lmonomial(p, Rem)

                 2                       2
        [poly(2 x  y, [x, y]), poly(3 x y  + 6, [x, y])]
```

The leading monomial and the reductum add up to the polynomial p:

```
>> p = q[1] + q[2]

        2           2
 poly(2 x  y + 3 x y  + 6, [x, y]) =

           2           2
    poly(2 x  y + 3 x y  + 6, [x, y])

>> delete p, q:
```

Example 4. We define a polynomial over the integers modulo 7:

```
>> p := poly(3*x + 4, [x], Dom::IntegerMod(7)): lmonomial(p)

                  poly(3 x, [x], Dom::IntegerMod(7))
```

This polynomial cannot be regarded as a polynomial with respect to another indeterminate, because the "coefficient" `3*x` cannot be interpreted as an element of the coefficient ring `Dom::IntegerMod(7)`:

```
>> lmonomial(p, [y])

                              FAIL

>> delete p:
```

Example 5. We demonstrate the evaluation strategy of `lmonomial`:

```
>> p := poly(6*x^6*y^2 + x^2 + 2, [x]): y := 4: lmonomial(p)

                              2   6
                       poly((6 y ) x , [x])
```

Evaluation is enforced by `eval`:

```
>> mapcoeffs(%, eval)

                                  6
                          poly(96 x , [x])

>> delete p, y:
```

`ln` – the natural logarithm

`ln(x)` represents the natural logarithm of x.

Call(s):

- `ln(x)`

Parameters:

`x` — an arithmetical expression or a floating point interval

Return Value: an arithmetical expression or a floating point interval

Overloadable by: `x`

Side Effects: When called with a floating point argument, the function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Related Functions: `dilog`, `log`, `polylog`

Details:

- The logarithm is defined for all complex arguments $x \neq 0$.
- For most exact arguments an unevaluated function call is returned. It is subject to some simplifications:

  - Arguments of the form $x = \exp(y)$ with y of the type `Type::Numeric` yield the result $\ln(\exp(y)) = y + k\,i\,2\,\pi$. Here k is some suitable integer, such that the imaginary part of the result lies in the interval $(-\pi, \pi]$. Similar simplifications occur for arguments of the form $x = \exp(y)^a$.
  - Negative integer and rational arguments x are rewritten according to $\ln(x) = i\,\pi + \ln(-x)$. Arguments of the form $x = 1/n$ with some integer n are rewritten according to $\ln(1/n) = -\ln(n)$.
  - The following special values are implemented: $\ln(1) = 0$, $\ln(-1) = i\,\pi$, $\ln(\pm\,i) = \pm\,i\,\pi/2$, $\ln(\texttt{infinity}) = \texttt{infinity}$, $\ln(-\texttt{infinity}) = i\,\pi + \texttt{infinity}$.

- Floating point results are computed for floating point arguments. The imaginary part of the result takes values in the interval $(-\pi, \pi]$. The negative real axis is a branch cut, the imaginary part of the result jumps when crossing the cut. On the negative real axis, the imaginary part is π according to $\ln(x) = i\,\pi + \ln(-x)$, $x < 0$. Cf. example 2.
- For floating point interval arguments (of type `DOM_INTERVAL`), the return value will be of type `DOM_INTERVAL`, properly rounded outwards. Note that this implies that the result does not contain complex (non-real) numbers. See example 5.
- Note that arithmetical rules such as $\ln(x\,y) = \ln(x) + \ln(y)$ are not valid throughout the complex plane. Use properties to mark identifiers as real and apply functions such as `expand`, `combine` or `simplify` to manipulate expressions involving `ln`. Cf. example 4.

Example 1. We demonstrate some calls with exact and symbolic input data:

```
>> ln(2), ln(-3), ln(1/4), ln(1 + I), ln(x^2)

                                                     2
         ln(2), I PI + ln(3), -ln(4), ln(1 + I), ln(x )
```

Floating point values are computed for floating point arguments:

```
>> ln(123.4), ln(5.6 + 7.8*I), ln(1.0/10^20)

     4.815431111, 2.261980065 + 0.948125538 I, -46.05170186
```

Some special symbolic simplifications are implemented:

```
>> ln(1), ln(-1), ln(exp(-5)), ln(exp(5 + 27/4*I))

               0, I PI, -5, (5 + 27/4 I) - 2 I PI
```

Example 2. The negative real axis is a branch cut. The imaginary part of the values returned by `ln` jump when crossing this cut:

```
>> ln(-2.0), ln(-2.0 + I/10^1000), ln(-2.0 - I/10^1000)

 0.6931471806 + 3.141592654 I, 0.6931471806 + 3.141592654 I,

    0.6931471806 - 3.141592654 I
```

Example 3. The functions `diff`, `float`, `limit`, `series` etc. handle expressions involving `ln`:

```
>> diff(ln(x^2), x), float(ln(PI + I))

                2
                -, 1.192985153 + 0.3081690711 I
                x

>> limit(ln(x)/x, x = infinity), series(x*ln(sin(x)), x = 0, 10)

                        3     5     7       9
                       x     x     x       x         11
          0, x ln(x) - -- - --- - ---- - ----- + O(x   )
                        6   180   2835   37800
```

Example 4. The functions `expand`, `combine`, and `simplify` react to properties set via `assume`. The following call does not produce an expanded result, because the arithmetical rule $\ln(x\,y) = \ln(x) + \ln(y)$ does not hold for arbitrary complex x, y:

```
>> expand(ln(x*y))

                              ln(x y)
```

However, the rule is valid, if one of the factors is real and positive:

```
>> assume(x > 0): expand(ln(x*y))

                           ln(x) + ln(y)

>> combine(%, ln)

                              ln(x y)

>> simplify(ln(x^3*y) - ln(x) - ln(y))

                              2 ln(x)

>> unassume(x):
```

Example 5. The logarithm of an interval is the image set of the logarithm function over the set represented by the interval:

```
>> ln(1 ... 2)

                -1.843143694e-18 ... 0.6931471806

>> ln(-1 ... 1)

  (RD_NINF ... 0.0) + (-1.875958165e-2525222 ... 3.141592654) I
```

This definition extends to unions of intervals:

```
>> ln(1 ... 2 union 3 ... 4)

 -1.843143694e-18 ... 0.6931471806 union

    1.098612288 ... 1.386294362
```

Changes:

- Floating point intervals are accepted.

`loadlib` – load a library package

`loadlib(libname)` loads the library package `libname`.

Call(s):

- `loadlib(libname)`

Parameters:

`libname` — the package name: a string

Return Value: `TRUE` if the package has been loaded successfully, and `FALSE` if the package was already loaded.

Related Functions: `export`, `LIBPATH`, `loadmod`, `loadproc`, `package`, `Pref::verboseRead`

Details:

- **`loadlib` is obsolete. Please use `package` instead.**
- `loadlib` loads the library package with the name `libname`. The library packages from the MuPAD distribution, such as, e.g., `fp`, are loaded automatically at startup. Thus `loadlib` is only relevant for loading user defined packages.
- `loadlib` searches for the initialization file of the given library package. This may be either a MuPAD binary file `libname.mb` or a MuPAD text file `libname.mu`, where `libname` is the name of the library package. The file is searched for in the subdirectory `LIBFILES` relative to each of the directories given by `LIBPATH`. `loadlib` first searches all corresponding directories for the binary file `libname.mb` and reads the first matching file. If none is found, then the text file `libname.mu` is tried.

  The file `fp.mu` in the subdirectory `LIBFILES` of the directory where the MuPAD system library is installed can be used as a model for a library initialization file.
- You may make the system automatically load user defined library packages by adding the local directory where the packages reside to `LIBPATH`. The initialization files for the library packages must be located in the subdirectory `LIBFILES` of the local directory.

- `loadlib` returns `TRUE` if the package was found and successfully loaded. `FALSE` is returned if the package is already loaded. An error occurs if the package was not found.
- A library is loaded only once at the first call of `loadlib`. A subsequent call does not re-load the same library.

`loadmod` – load a module

`loadmod("modulename")` loads the dynamic module named `modulename`.

`loadmod()` checks whether the MuPAD kernel supports dynamic modules.

Call(s):

- `loadmod("modulename")`
- `loadmod()`

Parameters:

`"modulename"` — the name of a module: a character string

Return Value: `loadmod()` returns `TRUE` or `FALSE`; `loadmod("modulename")` returns a module domain of type `DOM_DOMAIN`.

Side Effects: `loadmod("modulename")` assigns a value to the identifier `modulename`. E.g., after `loadmod("stdmod")`, the identifier `stdmod` has the loaded module as its value.

Further Documentation: Dynamic Modules - User's Manual and Programming Guide for MuPAD 1.4, Andreas Sorgatz, Oct 1998, Springer Verlag, Heidelberg, with CD-ROM, ISBN 3-540-65043-1.

Related Functions: `external`, `export`, `module::new`, `package`, `unloadmod`

Details:

- `loadmod()` returns `TRUE` if this MuPAD version supports the use of dynamic modules. Otherwise, it returns `FALSE`.
- `loadmod("modulename")` loads the dynamic module named `modulename`. Doing this, it defines a corresponding module domain, assigns it to the identifier `modulename` and returns the domain to the MuPAD session. A previously assigned value of the identifier `modulname` is overwritten.

- If the module domain already exists, it is overwritten and the warning 'Warning: Redefinition of domain ...' is displayed.
- The module file "modulename.mdm" is first searched for in the directories defined in the variable READPATH, then in the current working directory and, finally, in the MuPAD module directory.
- If the module cannot be loaded, the evaluation is aborted with an error message.
- If the file "modulename.mdg" exists, then it contains MuPAD objects that are likewise loaded and bound to the module domain. If an error occurs while loading these objects, a warning is displayed and MuPAD tries once more to load them at each call of the module functions affected by it.
- Apart from the module machine code file "modulename.mdm", there may also be a text file "modulename.mdh" containing a brief description of the module. This documentation can be read online using the module function modulename::doc() or modulename::doc("methodname"), respectively.
- loadmod is a function of the system kernel.

Example 1. The following call loads the dynamic module stdmod:

```
>> loadmod("stdmod")

                     stdmod

>> type(stdmod);

                   DOM_DOMAIN
```

Since modules are represented as domains, they can be used in the same way as library packages or other MuPAD domains. E.g., a module function is called with the prefix modulename:

```
>> stdmod::which("stdmod")

 "/usr/local/mupad/linux/modules/stdmod.mdm"
```

As for libraries, info can also be used to get information about a loaded module:

```
>> info(stdmod)

 Module: 'stdmod' created on 28.Sep.00 by mmg R-2.0.0
 Module: Extended Module Management

 -- Interface:
 stdmod::age,  stdmod::doc,   stdmod::help, stdmod::max,
 stdmod::stat, stdmod::which
```

The function `export` exports all public functions of the module. After this, the method `"which"` can be called without the domain prefix `stdmod`:

```
>> export(stdmod): which("stdmod")

 "/usr/local/mupad/linux/modules/stdmod.mdm"
```

Example 2. Documentation of a dynamic module named `modulename` may be provided by a plain text file "`modulename.mdh`" which must be located in the same directory as the module file "`modulename.mdm`". Such documentation can be accessed as demonstrated below. Cf. `module::help` for details.

```
>> stdmod::doc()

 MODULE
   stdmod - Extended Module Management

 INTRODUCTION
   This module provides functions for an extended module ...

 INTERFACE
   age, doc, help, max, stat, which
```

Above, the introductory page of the module documentation was displayed. Below, using the argument `"doc"`, the help page of the function `stdmod::doc` is shown:

```
>> stdmod::doc("doc")

 NAME
   stdmod::doc - Display online documentation

 SYNOPSIS
   ...

 SEE ALSO
   info, module::help
```

Background:

- The kernel functions `external`, `loadmod`, and `unloadmod` provide basic tools for accessing modules. Extended facilities are available with the `module` library.
- Only one instance of a dynamic module can exist in memory at time. Each further call of `loadmod` only reloads the machine code if it was unloaded or displaced before. However, the module domain is always re-created on loading.

- The machine code of dynamic modules can be unloaded during a MuPAD session using the function `unloadmod`.
- MuPAD provides a module resource management which may displace the machine code of dynamic modules if they are currently not needed, or if there is a lack of memory resources.
- Besides *dynamic* modules, MuPAD also supports so-called *static* modules which cannot be unloaded or displaced at runtime automatically. Also refer to `unloadmod`.

`loadproc` – load an object on demand

`loadproc` loads a MuPAD object from a file when it is first accessed.

Call(s):

- `object := loadproc(object, path, file)`

Parameters:

`object`	—	any MuPAD object that is a valid left hand side for an assignment
`path`	—	a relative path name with a terminating path separator: a string
`file`	—	a file name without suffix: a string

Return Value: an element of the domain `stdlib::LoadProc` (see "Background" below).

Related Functions: `export`, `finput`, `fread`, `LIBPATH`, `loadmod`, `package`, `pathname`, `Pref::verboseRead`, `read`

Details:

- The MuPAD library is quite big. However, users typically need only a small part of the library. It would be very time and memory consuming to load the whole library at startup. `loadproc` provides a concept for delaying the process of loading a predefined object, such as a library domain or a library procedure, until the time when it is first needed.
- `loadproc` returns an element of a special domain. This element only stores the information about the file where `object` is defined, but it does not yet read the file. This happens only when `object` is used for the first time.

The path and the name of the file are given by the two strings `path` and `file`, respectively. The function `pathname` is useful for creating path names in a platform independent way.

- When `object` is evaluated for the first time, the system first tries to read the MuPAD binary file

  `path . "." . file . ".mb"`,

  where `.` is the concatenation operator. MuPAD searches for this file relative to the directories given by `LIBPATH`. The first matching file is read. If the search fails, MuPAD tries the text file

  `path . "." . file . ".mu"`

  instead.

  The corresponding file must contain the "real" definition of `object`, typically a statement of the form `object := value`. If this is not the case, the system may run into infinite recursion.

  Finally, after the file has been read, `value` is returned as the value of `object`. The whole loading process is transparent to the user. See the example below for illustration.

- `loadproc` does not evaluate the first argument `object`, but the other arguments are evaluated as usual.

- To avoid side-effects, alias definitions are not in effect while the file is read, except those that are defined within the file. Alias definitions in the file are local to the file only; they are removed when the loading is finished.

Example 1. At system startup, the identifier `int` is initialized as follows:

```
>> int := loadproc(int, pathname("STDLIB"), "int"):
```

This tells the system that it finds the actual definition of the integration function `int` in the file `"STDLIB/int.mu"`, relative to the library path specified by `LIBPATH`, which by default points to MuPAD's installation directory.

When you first use `int`, e.g., by entering the command `int(t^2,t)`, MuPAD automatically loads the file `"STDLIB/int.mu"`. This file contains the following lines defining the actual function environment `int`:

```
int := funcenv(
proc(f, x = null())
begin
  if args(0) = 0 then error("No argument given") end_if;
  ...
end_proc):
```

After the file has been read, the function environment is returned as the value of `int`, and then the system proceeds as usual: the call `int(t^2,t)` is executed and its result `t^3/3` is returned.

Background:

- `loadproc` returns an object of the domain `stdlib::LoadProc`. This is an internal data type; manipulating its elements should never be necessary. Therefore it remains undocumented.

- Often a library source file provides definitions for several objects to be loaded via `loadproc`. In such a case, it may happen that an object is loaded even before it is first accessed, namely when another object is accessed whose definition is located in the same source file.

`log` – the logarithm to an arbitrary base

`log(b, x)` represents the logarithm of x to the base b.

Call(s):

- `log(b, x)`

Parameters:

`b` — either an identifier of domain type `DOM_IDENT` or a real numerical value of type `Type::Positive`.
`x` — an arithmetical expression

Return Value: an arithmetical expression.

Overloadable by: `x`

Side Effects: When called with a floating point argument, the function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Related Functions: `dilog`, `ln`, `polylog`

Details:

- Mathematically, $\log(b, x)$ coincides with $\ln(x)/\ln(b)$. Cf. example 3. The logarithm is defined for all complex arguments $x \neq 0$.

- The base b must be real, positive and not equal to 1. Internal simplifications are based on these assumptions.

- For most exact arguments an unevaluated function call is returned subject to some simplifications:

  - If $b = \exp(1)$, then $\log(b, x) = \ln(x)$ is returned.

- Mathematically, $\log(b, b^y) = y$ holds true for any real y. This simplification is implemented for the following cases: i) b is a symbolic identifier and y is of type `Type::Real`, ii) b is numerical and y is integer or rational.
- Negative integer and rational arguments x are rewritten according to $\log(b, x) = i\,\pi/\ln(b) + \log(b, -x)$. Rational arguments of the form $x = 1/n$ with some integer n are rewritten according to $\log(b, 1/n) = -\log(b, n)$.
- The following special values are implemented:

$$\log(b, 1) = 0\ ,\ \log(b, -1) = \frac{i\,\pi}{\ln(b)}\ ,\ \log(b, \pm\, i) = \pm\frac{i\,\pi}{2\,\ln(b)}.$$

Floating point results are computed if both arguments are numerical and at least one of them is a floating point number. The imaginary part of the result takes values in the interval $(-\pi/\ln(b), \pi/\ln(b)]$ for $b > 1$ (in the interval $[\pi/\ln(b), -\pi/\ln(b))$ for $b < 1$, respectively). The negative real axis is a branch cut, the imaginary part of the result jumps when crossing the cut. On the negative real axis, the imaginary part is $\pi/\ln(b)$ according to $\log(b, x) = i\,\pi/\ln(b) + \log(b, -x)$, $x < 0$. Cf. example 2.

Note that arithmetical rules such as $\log(b, x\,y) = \log(b, x) + \log(b, y)$ are not valid throughout the complex plane. Use properties to mark identifiers as real and apply functions such as `expand` or `simplify` to manipulate expressions involving `log`. Cf. example 5.

Example 1. We demonstrate some calls with exact and symbolic input data:

```
>> log(b, 2), log(2, 3), log(10, 10^2), log(10, 2*10^2),
   log(2, I), log(b, x^2)

                                             1/2 I PI          2
   log(b, 2), log(2, 3), 2, log(10, 200), --------, log(b, x )
                                               ln(2)
```

Note that the base may be a symbolic identifier. However, expressions are not accepted:

```
>> log(b + 1, 2)

 Error: base must be an identifier or of Type::Positive [log]

>> log(PI^2, 2)

 Error: base must be an identifier or of Type::Positive [log]
```

Floating point values are computed for floating point arguments:

```
>> log(2, 123.4), log(2.0, 5.6 + 7.8*I), log(10.0, 2/10^20)

       6.947198584, 3.263347423 + 1.367856012 I, -19.69897
```

Some special symbolic simplifications are implemented:

```
>> log(b, 1), log(b, -1), log(2/3, (4/9)^10), log(b, b^(-5))

                              I PI
                          0, -----, 20, -5
                             ln(b)
```

Example 2. The negative real axis is a branch cut. The imaginary part of the values returned by `log` jump when crossing this cut:

```
>> log(10, -2.0),
   log(10, -2.0 + I/10^1000),
   log(10, -2.0 - I/10^1000)

 0.3010299957 + 1.364376354 I, 0.3010299957 + 1.364376354 I,

    0.3010299957 - 1.364376354 I
```

Example 3. Use `rewrite` to rewrite `log` in terms of `ln`:

```
>> rewrite(log(b, x), ln), rewrite(log(10, 200), ln)

                        ln(x)  ln(200)
                        -----, -------
                        ln(b)   ln(10)
```

Example 4. The functions `diff`, `float`, `limit`, `series` etc. handle expressions involving `log`:

```
>> diff(log(b, x^2), x), float(log(10, PI + I))

             2
          -------, 0.5181068691 + 0.1338361271 I
          x ln(b)

>> limit(log(10, x)/x, x = infinity),
   series(x*log(x, sin(x)), x = 0)

                       3          5
                      x          x          7
            0, x - ------- - --------- + O(x )
                   6 ln(x)   180 ln(x)
```

Example 5. The functions `expand` and `simplify` react to properties set via `assume`. The following call does not produce an expanded result, because the arithmetical rule $\log(b, x\,y) = \log(b, x) + \log(b, y)$ does not hold for arbitrary complex x, y. Note, however, that `expand` rewrites `log` in terms of `ln`:

```
>> expand(log(10, x*y))

                          ln(x y)
                          -------
                           ln(10)
```

However, the rule is valid, if one of the factors is real and positive:

```
>> assume(x > 0): expand(log(b, x*y))

                        ln(x)   ln(y)
                        ----- + -----
                        ln(b)   ln(b)

>> simplify(log(b, x^3*y) - log(b, x) - log(b, y))

                         2 log(b, x)

>> unassume(x):
```

`lterm` – the leading term of a polynomial

`lterm(p)` returns the leading term of the polynomial `p`.

Call(s):

`lterm(p <, vars> <, order>)`

Parameters:

`p`	—	a polynomial of type `DOM_POLY` or a polynomial expression
`vars`	—	a list of indeterminates of the polynomial: typically, identifiers or indexed identifiers
`order`	—	the term ordering: either *LexOrder* or *DegreeOrder* or *DegInvLexOrder* or a user-defined term ordering of type `Dom::MonomOrdering`. The default is the lexicographical ordering *LexOrder*.

Return Value: a polynomial of the same type as `p`. An expression is returned if `p` is an expression. `FAIL` is returned if the input cannot be converted to a polynomial.

Overloadable by: `p`

Related Functions: `coeff`, `degree`, `degreevec`, `ground`, `lcoeff`, `ldegree`, `lmonomial`, `nterms`, `nthcoeff`, `nthmonomial`, `nthterm`, `poly`, `poly2list`, `tcoeff`

Details:

- The argument `p` can either be a polynomial expression, or a polynomial generated by `poly`, or an element of some polynomial domain overloading `lterm`.
- The identity `lterm`(p) `lcoeff`(p) = `lmonomial`(p) holds.
- If a list of indeterminates is provided, then `p` is regarded as a polynomial in these indeterminates. Note that the specified list does not have to coincide with the indeterminates of the input polynomial. Cf. example 1.
- The returned term is "leading" with respect to the lexicographical ordering, unless a different ordering is specified via the argument `order`. Cf. example 2.
- `lterm` returns `FAIL` if the input polynomial cannot be converted to a polynomial in the specified indeterminates. Cf. example 3.
- The leading term of the zero polynomial is the zero polynomial.
- For the orderings *`LexOrder`*, *`DegreeOrder`* and *`DegInvLexOrder`*, the result is computed by a fast kernel function. Other orderings are handled by slower library functions.

Example 1. We demonstrate how the indeterminates influence the result:

```
>> p := 2*x^2*y + 3*x*y^2 + 6:
   lterm(p), lterm(p, [x, y]), lterm(p, [y, x])

                                 2   2       2
                              x y , x  y, x y
```

Note that the indeterminates passed to `lterm` will be used, even if the polynomial provides different indeterminates :

```
>> p := poly(2*x^2*y + 3*x*y^2, [x, y]):
   lterm(p), lterm(p, [x, y]), lterm(p, [y, x]),
   lterm(p, [y]), lterm(p, [z])

       2                        2                        2
 poly(x  y, [x, y]), poly(x  y, [x, y]), poly(y  x, [y, x]),

         2
   poly(y , [y]), poly(1, [z])
```

```
>> delete p:
```

Example 2. We demonstrate how various orderings influence the result:

```
>> p := poly(5*x^4 + 4*x^3*y*z^2 + 3*x^2*y^3*z + 2, [x, y, z]):
   lterm(p), lterm(p, DegreeOrder), lterm(p, DegInvLexOrder)

       4                     3    2
 poly(x , [x, y, z]), poly(x  y z , [x, y, z]),

          2  3
    poly(x  y  z, [x, y, z])
```

The following call uses the reverse lexicographical order on 3 indeterminates:

```
>> lterm(p, Dom::MonomOrdering(RevLex(3)))

                          2  3
                    poly(x  y  z, [x, y, z])

>> delete p:
```

Example 3. We define a polynomial over the integers modulo 7:

```
>> p := poly(3*x + 4, [x], Dom::IntegerMod(7)): lterm(p)

                poly(x, [x], Dom::IntegerMod(7))
```

This polynomial cannot be regarded as a polynomial with respect to another indeterminate, because the "coefficient" `x` cannot be interpreted as an element of the coefficient ring `Dom::IntegerMod(7)`:

```
>> lterm(p, [y])

                              FAIL

>> delete p:
```

Example 4. The leading monomial is the product of the leading coefficient and the leading term:

```
>> p := poly(2*x^2*y + 3*x*y^2 + 6, [x, y]):
   mapcoeffs(lterm(p),lcoeff(p)) = lmonomial(p)
```

```
                 2                        2
      poly(2 x  y, [x, y]) = poly(2 x  y, [x, y])
```

```
>> delete p:
```

---

match – pattern matching

match(expression, pattern) checks whether the syntactical structure of expression matches pattern, and if so, returns a set of replacement equations transforming pattern into expression.

Call(s):

- match(expression, pattern <, option1, option2, ...>)

Parameters:

expression	—	a MuPAD expression
pattern	—	the pattern: a MuPAD expression
option1, option2, ...	—	optional arguments (see below)

Options:

Ass = {f1, f2, ...}	—	assume that the identifiers f1, f2, ... represent associative operators
Comm = {g1, g2, ...}	—	assume that the identifiers g1, g2, ... represent commutative operators
Cond = {p1, p2, ...}	—	conditional matching: consider only matches for which the conditions specified by the procedures p1, p2, ... are satisfied
Const = {c1, c2, ...}	—	assume that the identifiers c1, c2, ... represent constants
Null = {h1 = e1, h2 = e2, ...}	—	assume that the identifiers e1, e2, ... represent the neutral elements with respect to the operators h1, h2, ..., respectively

Return Value: a set of replacement equations or FAIL.

Related Functions: matchlib::analyze, simplify, subs, subsex, subsop

Details:

- `match` computes a set of replacement equations `S` for the identifiers occurring in `pattern`, such that `subs(pattern, S)` and `expression` coincide up to associativity, commutativity, and neutral elements.

- Most of the functionality of `match` is available via additional options. However, `match` is still in an experimental state, and some features may not work properly, yet.

- Without additional options, a purely syntactical matching is performed; associativity, commutativity, or neutral elements are not taken into account. In this case, `subs(pattern, S) = expression` holds for the set `S` of replacement equations returned by `match` if the matching was successful. Cf. example 1.

  You can declare these properties for operators via the options *`Ass`*, *`Comm`*, and *`Null`* (see below). Then `subs(pattern, S)` and `expression` need no longer be equal in MuPAD, but they can be transformed into each other by application of the rules implied by the options.

- Both `expression` and `pattern` may be arbitrary MuPAD expressions, i.e., both atomic expressions such as numbers, Boolean constants, and identifiers, and composite expressions.

- Each identifier without a value that occurs in `pattern`, including the 0th operands, is regarded as a *pattern variable*, in the sense that it may be replaced by some expression in order to transform `pattern` into `expression`. Use the option *`Const`* (see below) to declare identifiers as non-replaceable.

- With the exception of some automatic simplifications performed by the MuPAD kernel, distributivity is *not* taken into account. Cf. example 5.

- `match` evaluates its arguments, as usual. This evaluation usually encompasses a certain amount of simplification, which may change the syntactical structure of both `expression` and `pattern` in an unexpected way. Cf. example 6.

- Even if there are several possible matches, `match` returns at most one of them. Cf. example 7.

- If the structure of `expression` does not match `pattern`, `match` returns `FAIL`.

- If `expression` and `pattern` are equal, the empty set is returned.

- Otherwise, if a match is found and `expression` and `pattern` are different, then a set `S` of replacement equations is returned. For each pattern variable `x` occurring in `pattern` that is not declared constant via the option *`Const`*, `S` contains exactly one replacement equation of the form `x =`

`y`, and `y` is the expression to be substituted for `x` in order to transform `pattern` into `expression`.

Option <`Ass = {f1, f2, ...}`>**:**

- It is assumed that operators `f1, f2, ...` are associative and may take an arbitrary number of arguments, i.e., expressions such as `f1(f1(a, b), c)`, `f1(a, f1(b, c))`, and `f1(a, b, c)` are considered equal.

- No special rules for associative operators with less than two arguments apply. In particular, `f1(a)` and `a` are *not* considered equal.

Option <`Comm = {g1, g2, ...}`>**:**

- The operators `g1, g2, ...` are assumed to be commutative, i.e., expressions such as `g1(a, b)` and `g1(b, a)` are considered equal.

Option <`Cond = {p1, p2, ...}`>**:**

- Only matches satisfying the conditions specified by the procedures `p1, p2, ...` are considered. Each procedure must take exactly one argument and represents a condition on exactly one pattern variable. The name of the procedure's formal argument must be equal to the name of a pattern variable occurring in `pattern` that is not declared constant via the option *`Const`*. Each condition procedure must return an expression that the function `bool` can evaluate to one of the Boolean values `TRUE` or `FALSE`.

  Anonymous procedures created via `->` can be used to express simple conditions. Cf. example 8.

- If a possible match is found, given by a set of replacement equations `S`, then `match` checks whether all specified conditions are satisfied by calling `bool(p1(y1) and p2(y2) and ...)`, where `y1` is the expression to be substituted for the pattern variable `x1` that agrees with the formal argument of the procedure `p1`, etc. If the return value of the call is `TRUE`, then `match` returns `S`. Otherwise, the next possible match is tried.

  For example, if `p1` is a procedure with formal argument `x1`, where `x1` is a pattern variable occurring in `pattern`, then a match `S = {..., x1 = y1, ...}` is considered valid only if `bool(p1(y1))` returns `TRUE`.

- There can be at most one condition procedure for each pattern variable. If necessary, use the logical operators `and` and `or` as well as the control structures `if` and `case` to combine several conditions for the same pattern variable in one condition procedure. Cf. example 8.

Option <`Const = {c1, c2, ...}`>:

- The identifiers `c1, c2, ...` are regarded as constants, i.e., they must match literally and must not be replaced in order to transform `pattern` into `expression`.

Option <`Null = {h1 = e1, h2 = e2, ...}`>:

- It is assumed that `e1, e2, ...` are the neutral elements with respect to the associative operations `h1, h2, ...` i.e., expressions such as `h1(a, e1)`, `h1(e1, a)`, and `h1(a)` are considered equal.
- This declaration affects only operators that are declared associative via the option *Ass*. Moreover, the neutral elements are not implicitly assumed to be constants.

Example 1. All identifiers of the following pattern are pattern variables:

```
>> match(f(a, b), f(X, Y))

                    {X = a, Y = b, f = f}
```

The function `f` is declared non-replaceable:

```
>> match(f(a, b), f(X, Y), Const = {f})

                       {X = a, Y = b}
```

Example 2. The following call contains a condition for the pattern variable `X`:

```
>> match(f(a, b), f(X, Y), Const = {f}, Cond = {X -> not has(X, a)})

                            FAIL
```

If the function `f` is declared commutative, the expression matches the given pattern—in contrast to the preceding example:

```
>> match(f(a, b), f(X, Y), Const = {f}, Comm = {f},
                           Cond = {X -> not has(X, a)})

                       {X = b, Y = a}
```

Example 3. The following expression cannot be matched since the number of arguments of the expression and the pattern are different:

```
>> match(f(a, b, c), f(X, Y), Const = {f})

                        FAIL
```

We declare the function `f` associative with the option *`Ass`*. In this case the pattern matches the given expression:

```
>> match(f(a, b, c), f(X, Y), Const = {f}, Ass = {f})

                  {X = a, Y = f(b, c)}
```

Example 4. If, however, the function call in the pattern has more arguments than the corresponding function call in the expression, no match is found:

```
>> match(f(a, b), f(X, Y, Z), Const = {f}, Ass = {f})

                        FAIL
```

If the neutral element with respect to the operator `f` is known, additional matches are possible by substituting it for some of the pattern variables:

```
>> match(f(a, b), f(X, Y, Z), Const = {f}, Ass = {f}, Null = {f = 0})

                  {X = a, Y = b, Z = 0}
```

Example 5. Distributivity is *not* taken into account in general:

```
>> match(a*x + a*y, a*(X + Y), Const = {a})

                        FAIL
```

The next call finds a match, but not the expected one:

```
>> match(a*(x + y), X + Y)

                 {X = a (x + y), Y = 0}
```

The following declarations and conditions do not lead to the expected result, either:

```
>> match(a*(x + y), a*X + a*Y, Const = {a},
                               Cond = {X -> X <> 0, Y -> Y <> 0})

                        FAIL
```

Example 6. Automatic simplifications can "destroy" the structure of the given expression or pattern:

```
>> match(sin(-2), sin(X))
```

```
FAIL
```

The result is `FAIL`, because the first argument `sin(-2)` is evaluated:

```
>> sin(-2)
```

```
-sin(2)
```

You can circumvent this problem by using `hold`:

```
>> match(hold(sin(-2)), sin(X))
```

```
{X = -2}
```

Example 7. `match` returns only one possible match:

```
>> match(a + b + c + 1, X + Y)
```

```
{X = a, Y = b + c + 1}
```

To obtain other solutions, use conditions to exclude the solutions that you already have:

```
>> match(a + b + c + 1, X + Y, Cond = {X <> a})
```

```
{Y = a, X = b + c + 1}
```

```
>> match(a + b + c + 1, X + Y, Cond = {X <> a and Y <> a})
```

```
{X = b, Y = a + c + 1}
```

```
>> match(a + b + c + 1, X + Y,
         Cond = {X <> a and X <> b and Y <> a})
```

```
{X = c, Y = a + b + 1}
```

Example 8. Every pattern variable can have at most one condition procedure. Simple conditions can be given by anonymous procedures (`->`):

```
>> match(a + b, X + Y, Cond = {X -> X <> a, Y -> Y <> b})
```

```
{X = b, Y = a}
```

Several conditions on a pattern variable can be combined in one procedure:

```
>> Xcond := proc(X) begin
     if domtype(X) = DOM_IDENT then
       X <> a and X <> b
     else
       X <> 0
     end_if
   end_proc:
```

```
>> match(sin(a*b), sin(X*Y), Cond = {Xcond})
```

```
                    {X = a b, Y = 1}
```

```
>> match(sin(a*c), sin(X*Y), Cond = {Xcond})
```

```
                    {Y = a, X = c}
```

```
>> delete Xcond:
```

`matrix` – create a matrix or a vector

`matrix(m, n, [[a11, a12, ...], [a21, a22, ...], ...])` returns the $m \times n$ matrix

$$\begin{pmatrix} a_{11} & a_{12} & \cdots \\ a_{21} & a_{22} & \cdots \\ \vdots & \vdots & \vdots \end{pmatrix}.$$

`matrix(n, 1, [a1, a2, ...])` returns the $n \times 1$ column vector

$$\begin{pmatrix} a_1 \\ a_2 \\ \vdots \end{pmatrix}.$$

`matrix(1, n, [a1, a2, ...])` returns the $1 \times n$ row vector

$$\begin{pmatrix} a_1 & a_2 & \cdots \end{pmatrix}.$$

Call(s):

- `matrix(ListOfRows)`
- `matrix(List)`
- `matrix(Array)`

- `matrix(Matrix)`
- `matrix(m, n)`
- `matrix(m, n, ListOfRows)`
- `matrix(1, n, List)`
- `matrix(m, 1, List)`
- `matrix(m, n, List,` *`Diagonal`*`)`
- `matrix(m, n, List,` *`Banded`*`)`
- `matrix(m, n, f)`
- `matrix(m, n, g,` *`Diagonal`*`)`

Parameters:

`ListOfRows`	—	a nested list of at most `m` rows, each row being a list with at most `n` elements
`Array`	—	a one- or two-dimensional array
`Matrix`	—	a matrix, i.e., an object of a data type of category `Cat::Matrix`
`m`	—	the number of rows: a positive integer
`n`	—	the number of columns: a positive integer
`List`	—	a list
`f`	—	a function or a functional expression of two arguments
`g`	—	a function or a functional expression of one argument

Options:

`Diagonal`	—	create a diagonal matrix
`Banded`	—	create a banded Toeplitz matrix

Return Value: a matrix of the domain type `Dom::Matrix()`.

Related Functions: `array`, `DOM_ARRAY`, `Dom::Matrix`, `Dom::SparseMatrix`, `sparsematrix`

Details:

- `matrix` creates matrices and vectors. A vector with n entries is either an $n \times 1$ matrix (a column vector) or a $1 \times n$ matrix (a row vector).

  Matrix and vector components must be arithmetical expressions. For specific component domains, refer to the help page of `Dom::Matrix`.

- 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 `matrix`, then `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. Otherwise, `FAIL` is returned.

See example 1.

- Many system functions accept matrices as input, such as `map`, `subs`, `has`, `zip`, `conjugate` to compute the complex conjugate of a matrix, `norm` to compute matrix norms, or even `exp` to compute the exponential of a matrix. See example 4.

- 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` generated by `matrix`, call `linalg::det(A)`. The command `linalg::gaussJordan(A)` performs Gauss-Jordan elimination on `A` to transform `A` to its reduced row echelon form. Cf. example 2.

  See the help page of `linalg` for a list of available functions of this package.

- `matrix` is an abbreviation for the domain `Dom::Matrix()`. You find more information about this data type for matrices on the corresponding help page.

- Matrix components can be extracted by the usual index operator `[ ]`, which also works for lists, arrays, and tables. The call `A[i, j]` extracts the matrix component in the ith row and the jth column.

  Assignments to matrix components are performed similarly. The call `A[i, j] := c` replaces the matrix component in the ith row and the jth column of A by `c`.

  If one of the indices is not in its valid range, then an error message is issued.

  The index operator also extracts submatrices. The call `A[r1..r2, c1..c2]` creates the submatrix of A comprising the rows with the indices $r_1, r_1 + 1, \ldots, r_2$ and the columns with the indices $c_1, c_1 + 1, \ldots, c_2$ of A.

  See examples 3 and 5.

- `matrix(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. See example 6.

  It might be a good idea first to create a two-dimensional array from that list before calling `matrix`. This is due to the fact that creating a matrix from an array is the fastest way one can achieve. However, in this case the sublists must have the same number of elements.

- `matrix(List)` creates an $m \times 1$ column vector with components taken from the nonempty list, where m is the number of entries of `List`. See example 5.

- `matrix(Array)` or `matrix(Matrix)` create a new matrix with the same dimension and the components of `Array` or `Matrix`, respectively. The array must not contain any uninitialized entries. If `Array` is one-dimensional, then the result is a column vector. Cf. example 7.

- The call `matrix(m, n)` returns the $m \times n$ zero matrix.

- `matrix(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. See example 6.

- `matrix(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. See example 5.

- `matrix(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. See example 5.

- `matrix(m, n, f)` returns the matrix whose (i, j)th component is `f(i,j)`. The row index i runs from 1 to m and the column index j from 1 to n. See example 8.

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.

- `matrix(m, n, List, `*`Diagonal`*`)` creates the $m \times n$ diagonal matrix whose diagonal elements are the entries of `List`; see example 9.

  `List` must have at most $\min(m, n)$ entries. If it has fewer elements, then the remaining diagonal elements are set to zero.

- `matrix(m, n, g, `*`Diagonal`*`)` returns the matrix whose ith diagonal element is `g(i)`, where the index i runs from 1 to $\min(m, n)$. See example 9.

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.

- `matrix(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 bandwidth of the resulting matrix is at most h.

  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.

  See example 10.

Example 1. We create the 2×2 matrix

$$\begin{pmatrix} 1 & 5 \\ 2 & 3 \end{pmatrix}$$

by passing a list of two rows to `matrix`, where each row is a list of two elements, as follows:

```
>> A := matrix([[1, 5], [2, 3]])

                          +-      -+
                          |  1, 5  |
                          |        |
                          |  2, 3  |
                          +-      -+
```

In the same way, we generate the following 2×3 matrix:

```
>> B := matrix([[-1, 5/2, 3], [1/3, 0, 2/5]])

                       +-                 -+
                       |   -1, 5/2,   3    |
                       |                   |
                       |  1/3,   0,  2/5   |
                       +-                 -+
```

We can do matrix arithmetic using the standard arithmetical operators of MuPAD. For example, the matrix product $A \cdot B$, the 4th power of A, and the scalar multiplication of A by $\frac{1}{3}$ are given by:

```
>> 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   |
+-                 -+  +-           -+  +-          -+
```

Since the dimensions of the matrices A and B differ, the sum of A and B is not defined and MuPAD returns an error message:

```
>> A + B
```

```
Error: dimensions don't match [(Dom::Matrix(Dom::ExpressionFie\
ld()))::_plus]
```

To compute the inverse of A, enter:

```
>> 1/A
```

```
+-              -+
|  -3/7,  5/7   |
|                |
|   2/7, -1/7   |
+-              -+
```

If a matrix is not invertible, then the result of this operation is `FAIL`:

```
>> C := matrix([[2, 0], [0, 0]])
```

```
+-      -+
|  2, 0  |
|        |
|  0, 0  |
+-      -+
```

```
>> C^(-1)
```

```
FAIL
```

Example 2. In addition to standard matrix arithmetic, the library `linalg` offers a lot of functions handling matrices. For example, the function `linalg::rank` determines the rank of a matrix:

```
>> A := matrix([[1, 5], [2, 3]])
```

```
+-      -+
|  1, 5  |
|        |
|  2, 3  |
+-      -+
```

```
>> linalg::rank(A)
```

```
                                2
```

The function `linalg::eigenvectors` computes the eigenvalues and the eigenvectors of `A`:

```
>> linalg::eigenvectors(A)

 -- --                  -- +-              -+ -- --
 |  |                   |  |    1/2         |  |  |
 |  |                   |  |  11            |  |  |
 |  |    1/2            |  |  ----- - 1/2   |  |  |
 |  |  11     + 2, 1,   |  |    2           |  |  |,
 |  |                   |  |                |  |  |
 |  |                   |  |       1        |  |  |
 -- --                  -- +-              -+ -- --

    --                  -- +-                 -+ -- -- --
    |                   |  |      1/2          |  |  |  |
    |                   |  |    11             |  |  |  |
    |         1/2       |  |  - ----- - 1/2    |  |  |  |
    |  2 - 11     , 1,  |  |      2            |  |  |  |
    |                   |  |                   |  |  |  |
    |                   |  |        1          |  |  |  |
    --                  -- +-                 -+ -- -- --
```

To determine the dimension of a matrix use the function `linalg::matdim`:

```
>> linalg::matdim(A)

                             [2, 2]
```

The result is a list of two positive integers, the row and column number of the matrix.

Use `info(linalg)` to obtain a list of available functions, or enter `?linalg` for details about this library.

Example 3. Matrix entries can be accessed with the index operator `[ ]`:

```
>> 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   |
                          +-              -+
```

```
>> A[2, 1] * A[1, 2] - A[3, 1] * A[1, 3]

                                 7
```

You can redefine a matrix entry by assigning a value to it:

```
>> A[1, 2] := a^2: A

                         +-              -+
                         |        2       |
                         |   1, a , 3, 4  |
                         |                |
                         |   2,  0, 4, 1  |
                         |                |
                         |  -1,  0, 5, 2  |
                         +-              -+
```

The index operator can also be used to extract submatrices. The following call creates a copy of the submatrix of A comprising the second and the third row and the first three columns of A:

```
>> A[2..3, 1..3]

                           +-           -+
                           |   2, 0, 4   |
                           |             |
                           |  -1, 0, 5   |
                           +-           -+
```

The index operator does *not* allow to replace a submatrix of a given matrix by another matrix. Use `linalg::substitute` to achieve this.

Example 4. Some system functions can be applied to matrices. For example, if you have a matrix with symbolic entries and want to have all entries in expanded form, simply apply the function `expand`:

```
>> 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)   |
                   +-                             -+
```

```
>> expand(A)

                  +-                                -+
                  |               2    2    2    2   |
                  |   - 2 a b + a   + b , a   + b    |
                  |                                  |
                  |         2    2         2    2    |
                  |        a  + b ,       a  - b     |
                  +-                                -+
```

You can differentiate all matrix components with respect to some indeterminate:

```
>> diff(A, a)

                        +-                 -+
                        |  2 a - 2 b, 2 a   |
                        |                   |
                        |     2 a,    2 a   |
                        +-                 -+
```

The following command evaluates all matrix components at a given point:

```
>> subs(A, a = 1, b = -1)

                             +-       -+
                             |  4, 2   |
                             |         |
                             |  2, 0   |
                             +-       -+
```

Note that the function subs does not evaluate the result of the substitution. For example, we define the following matrix:

```
>> A := matrix([[sin(x), x], [x, cos(x)]])

                        +-                 -+
                        |  sin(x),    x     |
                        |                   |
                        |     x,   cos(x)   |
                        +-                 -+
```

Then we substitute $x = 0$ in each matrix component:

```
>> B := subs(A, x = 0)

                        +-                 -+
                        |  sin(0),    0     |
                        |                   |
                        |     0,   cos(0)   |
                        +-                 -+
```

You see that the matrix components are not evaluated completely: for example, if you enter `sin(0)` directly, it evaluates to zero.

The function `eval` can be used to evaluate the result of the function `subs`. However, `eval` does not operate on matrices directly, and you must use the function `map` to apply the function `eval` to each matrix component:

```
>> map(B, eval)

                          +-       -+
                          |  0, 0   |
                          |         |
                          |  0, 1   |
                          +-       -+
```

The function `zip` can 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   |
                          +-         -+
```

Example 5. A vector is either an $m \times 1$ matrix (a column vector) or a $1 \times n$ matrix (a row vector). To create a vector with `matrix`, pass the dimension of the vector and a list of vector components as argument to `matrix`:

```
>> row_vector    := matrix(1, 3, [1, 2, 3]);
   column_vector := matrix(3, 1, [1, 2, 3])

                          +-         -+
                          | 1, 2, 3   |
                          +-         -+

                            +-   -+
                            |  1  |
                            |     |
                            |  2  |
                            |     |
                            |  3  |
                            +-   -+
```

If the only argument of `matrix` is a non-nested list or a one-dimensional array, then the result is a column vector:

```
>> matrix([1, 2, 3])
```

```
+-   -+
|  1  |
|     |
|  2  |
|     |
|  3  |
+-   -+
```

For a row vector `r`, the calls `r[1, i]` and `r[i]` both return the ith vector component of `r`. Similarly, for a column vector `c`, the calls `c[i, 1]` and `c[i]` both return the ith vector component of `c`.

For example, to extract the second component of the vectors `row_vector` and `column_vector`, we enter:

```
>> row_vector[2], column_vector[2]
```

```
2, 2
```

Use the function `linalg::vecdim` to determine the number of components of a vector:

```
>> linalg::vecdim(row_vector), linalg::vecdim(column_vector)
```

```
3, 3
```

The number of components of a vector can also be determined directly by the call `nops(vector)`.

The dimension of a vector can be determined as described above in the case of matrices:

```
>> linalg::matdim(row_vector),
   linalg::matdim(column_vector)
```

```
[1, 3], [3, 1]
```

See the `linalg` package for functions working with vectors, and the help page of `norm` for computing vector norms.

Example 6. In the following examples, we illustrate various calls of `matrix` as described above. We start by passing a nested list to `matrix`, where each inner list corresponds to a row of the matrix:

```
>> matrix([[1, 2], [2]])
```

```
+-      -+
|  1, 2  |
|        |
|  2, 0  |
+-      -+
```

The number of rows of the created matrix is the number of inner lists, namely $m = 2$. The number of columns is determined by the maximal number of entries of an inner list. In the example above, the first list is the longest one, and hence $n = 2$. The second list has only one element, and therefore the second entry in the second row of the returned matrix was set to zero.

In the following call, we use the same nested list, but in addition pass two dimension parameters to create a 4×4 matrix:

```
>> matrix(4, 4, [[1, 2], [2]])

                         +-            -+
                         |  1, 2, 0, 0  |
                         |              |
                         |  2, 0, 0, 0  |
                         |              |
                         |  0, 0, 0, 0  |
                         |              |
                         |  0, 0, 0, 0  |
                         +-            -+
```

In this case, the dimension of the matrix is given by the dimension parameters. As before, missing entries in an inner list correspond to zero, and in addition missing rows are treated as zero rows.

Example 7. A one- or two-dimensional array of arithmetical expressions, such as:

```
>> a := array(1..3, 2..4,
     [[1, 1/3, 0], [-2, 3/5, 1/2], [-3/2, 0, -1]]
   )

                      +-                -+
                      |     1,  1/3,  0  |
                      |                  |
                      |    -2,  3/5, 1/2 |
                      |                  |
                      |  -3/2,   0,   -1 |
                      +-                -+
```

can be converted into a matrix as follows:

```
>> A := matrix(a)

                      +-                -+
                      |     1,  1/3,  0  |
                      |                  |
                      |    -2,  3/5, 1/2 |
                      |                  |
                      |  -3/2,   0,   -1 |
                      +-                -+
```

Arrays serve, for example, as an efficient structured data type for programming. However, arrays do not have any algebraic meaning, and no mathematical operations are defined for them. If you convert an array into a matrix, you can use the full functionality defined for matrices as described above. For example, let us compute the matrix $2A - A^2$ and the Frobenius norm of A:

```
>> 2*A - A^2, norm(A, Frobenius)

            +-                   -+
            |   5/3,   2/15,  -1/6  |     1/2     1/2
            |                       |  450    4037
            |  -1/20, 113/75,  6/5  |, --------------
            |                       |        450
            |    -3,    1/2,   -3   |
            +-                   -+
```

Note that an array may contain uninitialized entries:

```
>> b := array(1..4): b[1] := 2: b[4] := 0: b

                      +-                 -+
                      | 2, ?[2], ?[3], 0 |
                      +-                 -+
```

matrix cannot handle arrays that have uninitialized entries, and responds with an error message:

```
>> matrix(b)

 Error: unable to define matrix over Dom::ExpressionField() [(D\
 om::Matrix(Dom::ExpressionField()))::new]
```

We initialize the remaining entries of the array b and convert it into a matrix, or more precisely, into a column vector:

```
>> b[2] := 0: b[3] := -1: matrix(b)

                            +-    -+
                            |   2  |
                            |      |
                            |   0  |
                            |      |
                            |  -1  |
                            |      |
                            |   0  |
                            +-    -+
```

Example 8. We show how to create a matrix whose components are defined by a function of the row and the column index. The entry in the ith row and the jth column of a Hilbert matrix (see also `linalg::hilbert`) is $1/(i+j-1)$. Thus the following command creates a 2×2 Hilbert matrix:

```
>> matrix(2, 2, (i, j) -> 1/(i + j - 1))

                                     +-           -+
                                     |   1,  1/2   |
                                     |             |
                                     |  1/2, 1/3   |
                                     +-           -+
```

The following two calls produce different results. In the first call, `x` is regarded as an unknown function, while it is a constant in the second call:

```
>> delete x:
   matrix(2, 2, x), matrix(2, 2, (i, j) -> x)

                +-                    -+   +-        -+
                |  x(1, 1), x(1, 2)   |   |  x, x   |
                |                      |,  |          |
                |  x(2, 1), x(2, 2)   |   |  x, x   |
                +-                    -+   +-        -+
```

Example 9. Diagonal matrices can be created by passing the option *Diagonal* and a list of diagonal entries:

```
>> matrix(3, 4, [1, 2, 3], Diagonal)

                                   +-             -+
                                   |  1, 0, 0, 0   |
                                   |               |
                                   |  0, 2, 0, 0   |
                                   |               |
                                   |  0, 0, 3, 0   |
                                   +-             -+
```

Hence, you can generate the 3×3 identity matrix as follows:

```
>> matrix(3, 3, [1 $ 3], Diagonal)

                                     +-          -+
                                     |  1, 0, 0   |
                                     |            |
                                     |  0, 1, 0   |
                                     |            |
                                     |  0, 0, 1   |
                                     +-          -+
```

Equivalently, you can use a function of one argument:

```
>> matrix(3, 3, i -> 1, Diagonal)

                          +-         -+
                          |  1, 0, 0  |
                          |           |
                          |  0, 1, 0  |
                          |           |
                          |  0, 0, 1  |
                          +-         -+
```

Since the integer `1` also represents a constant function, the following shorter call creates the same matrix:

```
>> matrix(3, 3, 1, Diagonal)

                          +-         -+
                          |  1, 0, 0  |
                          |           |
                          |  0, 1, 0  |
                          |           |
                          |  0, 0, 1  |
                          +-         -+
```

Example 10. Banded Toeplitz matrices (see above) can be created with the option *Banded*. The following command creates a matrix of bandwidth 3 with all main diagonal entries equal to 2 and all entries on the first sub- and super-diagonal equal to -1:

```
>> matrix(4, 4, [-1, 2, -1], Banded)

                      +-                 -+
                      |   2, -1,  0,  0   |
                      |                   |
                      |  -1,  2, -1,  0   |
                      |                   |
                      |   0, -1,  2, -1   |
                      |                   |
                      |   0,  0, -1,  2   |
                      +-                 -+
```

`map` – apply a function to all operands of an object

`map(object, f)` applies the function `f` to all operands of `object`.

Call(s):

- `map(object, f <, p1, p2, ...>)`

Parameters:

`object`	—	an arbitrary MuPAD object
`f`	—	a function
`p1, p2, ...`	—	any MuPAD objects accepted by `f` as additional parameters

Return Value: a copy of `object` with `f` applied to all operands.

Overloadable by: `object`

Related Functions: `eval`, `mapcoeffs`, `misc::maprec`, `op`, `select`, `split`, `subs`, `subsex`, `subsop`, `zip`

Details:

- `map(object, f)` returns a copy of `object` where each operand `x` has been replaced by `f(x)`. The object itself is not modified by `map` (see example 2).
- The second argument `f` may be a procedure generated via `->` or `proc` (e.g., `x -> x^2 + 1`), a function environment (e.g., `sin`), or a functional expression (e.g., `sin@exp + 2*id`).
- If optional arguments are present, then each operand `x` of `object` is replaced by `f(x, p1, p2, ...)` (see example 1).
- It is possible to apply an operator, such as `+` or `*`, to all operands of `object`, by using its functional equivalent, such as `_plus` or `_mult`. See example 1.
- In contrast to `op`, `map` does not decompose rational numbers and complex numbers further. Thus, if the argument is a rational number or a complex number, then `f` is applied to the number itself and not to the numerator and the denominator or the real part and the imaginary part, respectively (see example 3).
- If `object` is a string, then `f` is applied to the string as a whole and not to the individual characters (see example 3).
- If `object` is an expression, then `f` is applied to the operands of `f` as returned by `op` (see example 1).
- If `object` is an expression sequence, then this sequence is not flattened by `map` (see example 4).
- If `object` is a polynomial, then `f` is applied to the polynomial itself and not to all of its coefficients. Use `mapcoeffs` to achieve the latter (see example 3).

- If `object` is a list, a set, or an array, then the function `f` is applied to all elements of the corresponding data structure.
- If `object` is a table, the function `f` is applied to all *entries* of the table, not to the indices (see example 9). The entries are the right sides of the operands of a table.
- If `object` is an element of a library domain, then the slot `"map"` of the domain is called and the result is returned. This can be used to extend the functionality of `map` to user-defined domains. If no `"map"` slot exists, then `f` is applied to the object itself (see example 10).
- `map` does not evaluate its result after the replacement; use `eval` to achieve this. Nevertheless, internal simplifications occur after the replacement (see example 8).
- `map` does not descend recursively into an object; the function `f` is only applied to the operands at first level. Use `misc::maprec` for a recursive version of `map` (see example 11).
- `map` is a function of the system kernel.

Example 1. `map` works for expressions:

```
>> map(a + b + 3, sin)

                    sin(a) + sin(b) + sin(3)
```

The optional arguments of `map` are passed to the function being mapped:

```
>> map(a + b + 3, f, x, y)

              f(a, x, y) + f(b, x, y) + f(3, x, y)
```

In the following example, we add `10` to each element of a list:

```
>> map([1, x, 2, y, 3, z], _plus, 10)

              [11, x + 10, 12, y + 10, 13, z + 10]
```

Example 2. Like most other MuPAD functions, `map` does not modify its first argument, but returns a modified copy:

```
>> a := [0, PI/2, PI, 3*PI/2]:
   map(a, sin)

                         [0, 1, 0, -1]
```

The list `a` still has its original value:

```
>> a
```

```
--    PI       3 PI --
|  0, --, PI, ----  |
--     2        2  --
```

Example 3. `map` does not decompose rational and complex numbers:

```
>> map(3/4, _plus, 1), map(3 + 4*I, _plus, 1)
```

```
7/4, 4 + 4 I
```

`map` does not decompose strings:

```
>> map("MuPAD", text2expr)
```

```
MuPAD
```

`map` does not decompose polynomials:

```
>> map(poly(x^2 + x + 1), _plus, 1)
```

```
            2
poly(x  + x + 1, [x]) + 1
```

Use `mapcoeffs` to apply a function to all coefficients of a polynomial:

```
>> mapcoeffs(poly(x^2 + x + 1), _plus, 1)
```

```
              2
poly(2 x  + 2 x + 2, [x])
```

Example 4. The first argument is not flattened:

```
>> map((1, 2, 3), _plus, 2)
```

```
3, 4, 5
```

Example 5. Sometimes a MuPAD function returns a set or a list of big symbolic expressions containing mathematical constants etc. To get a better intuition about the result, you can map the function `float` to all elements, which often drastically reduces the size of the expressions:

```
>> solve(x^4 + x^2 + PI, x)

 {    1/2             1/2     1/2   1/2             1/2     1/2
 {   2    ((1 - 4 PI)    - 1)      2    ((1 - 4 PI)    - 1)
 { - --------------------------, --------------------------,
 {               2                             2

       1/2              1/2     1/2
      2    (- (1 - 4 PI)    - 1)
    - ----------------------------,
                    2

     1/2              1/2     1/2 }
    2    (- (1 - 4 PI)    - 1)    }
    ----------------------------- }
                  2               }

>> map(%, float)

 {- 0.7976383425 - 1.065939457 I,

    - 0.7976383425 + 1.065939457 I,

    0.7976383425 - 1.065939457 I, 0.7976383425 + 1.065939457 I}
```

Example 6. In the following example, we delete the values of all global identifiers in the current MuPAD session. The command `anames(All, User)` returns a set with the names of all user-defined global identifiers having a value. Mapping the function `_delete` to this set deletes the values of all these identifiers. Since the return value of `_delete` is the empty sequence `null()`, the result of the call is the empty set:

```
>> x := 3: y := 5: x + y

                                 8

>> map(anames(All, User), _delete)

                                {}

>> x + y

                               x + y
```

Example 7. It is possible to perform arbitrary actions with all elements of a data structure via a single `map` call. This works by passing an anonymous procedure as the second argument `f`. In the following example, we check that the fact "an integer $n \geq 2$ is prime if and only if $\varphi(n) = n - 1$", where φ denotes Euler's totient function, holds for all integer $2 \leq n < 10$. We do this by comparing the result of `isprime(n)` with the truth value of the equation $\varphi(n) = n - 1$ for all elements `n` of a list containing the integers between 2 and 9:

```
>> map([2, 3, 4, 5, 6, 7, 8, 9],
       n -> bool(isprime(n) = bool(numlib::phi(n) = n - 1)))

         [TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE, TRUE]
```

Example 8. The result of `map` is not evaluated further. If desired, you must request evaluation explicitly by `eval`:

```
>> map(sin(5), float);
   eval(%)

                              sin(5.0)

                            -0.9589242747
```

```
>> delete a:
   A := array(1..1, [a]);
   a := 0:
   map(A, sin);
   map(A, eval@sin);
   delete a:

                                +-   -+
                                | a |
                                +-   -+

                             +-        -+
                             | sin(a) |
                             +-        -+

                                +-   -+
                                | 0 |
                                +-   -+
```

Nevertheless, certain internal simplifications take place, such as the calculation of arithmetical operations with numerical arguments. The following call replaces `sqrt(2)` and `PI` by floating point approximations, and the system automatically simplifies the resulting sum:

```
>> map(sin(5) + cos(5), float)

                              -0.6752620892
```

Example 9. `map` applied to a table changes only the right sides (the entries) of each operand of the table. Assume the entries stand for net prices and the sales tax (16 percent in this case) must be added:

```
>> T := table(1 = 65, 2 = 28, 3 = 42):
   map(T, _mult, 1.16)

                              table(
                                3 = 48.72,
                                2 = 32.48,
                                1 = 75.4
                              )
```

Example 10. `map` can be overloaded for elements of library domains, if a slot `"map"` is defined. In this example `d` is a domain, its elements contains two integer numbers: an index and an entry (like a table). For nice input and printing elements of this domain the slots `"new"` and `"print"` are defined:

```
>> d := newDomain("d"):
   d::new := () -> new(d, args()):
   d::print := object -> _equal(extop(object)):
   d(1, 65), d(2, 28), d(3, 42)

                         1 = 65, 2 = 28, 3 = 42
```

Without a slot `"map"` the function `f` will be applied to the domain element itself. Because the domain `d` has no slot `"_mult"`, the result is the symbolic `_mult` call:

```
>> map(d(1, 65), _mult, 1.16),
   type(map(d(1, 65), _mult, 1.16))

                         1.16 (1 = 65), "_mult"
```

The slot `"map"` of this domain should map the given function only onto the second operand of a domain element. The domain `d` gets a slot `"map"` and `map` works properly (in the authors sense) with elements of this domain:

```
>> d::map := proc(obj, f)
             begin
               if args(0) > 2 then
                 d(extop(obj, 1), f(extop(obj, 2), args(3..args(0))))
               else
                 d(extop(obj, 1), f(extop(obj, 2)))
               end_if
             end_proc:
   map(d(1, 65), _mult, 1.16),
   map(d(2, 28), _mult, 1.16),
   map(d(3, 42), _mult, 1.16)
```

```
                 1 = 75.4, 2 = 32.48, 3 = 48.72
```

Example 11. `map` does not work recursively. Suppose that we want to de-nest a nested list. We use `map` to apply the function `op`, which replaces a list by the sequence of its operands, to all entries of the list `l`. However, this only affects the entries at the first level:

```
>> l := [1, [2, [3]], [4, [5]]]:
   map(l, op)
```

```
                       [1, 2, [3], 4, [5]]
```

Use `misc::maprec` to achieve the desired behavior:

```
>> [misc::maprec(l, {DOM_LIST} = op)]
```

```
                         [1, 2, 3, 4, 5]
```

`mapcoeffs` – apply a function to the coefficients of a polynomial

`mapcoeffs(p, F, a1, a2, ...)` applies the function `F` to the polynomial `p` by replacing each coefficient `c` in `p` by `F(c, a1, a2, ...)`.

Call(s):

- `mapcoeffs(p, F <, a1, a2, ...>)`
- `mapcoeffs(f, <vars, > F <, a1, a2, ...>)`

Parameters:

`p`	—	a polynomial of type `DOM_POLY`
`F`	—	a procedure
`a1, a2, ...`	—	additional parameters for the function `F`
`f`	—	a polynomial expression
`vars`	—	a list of indeterminates of the polynomial: typically, identifiers or indexed identifiers

Return Value: a polynomial of type `DOM_POLY`, or a polynomial expression, or `FAIL`.

Overloadable by: `p, f`

Related Functions: `coeff`, `degree`, `degreevec`, `lcoeff`, `ldegree`, `lterm`, `map`, `nterms`, `nthcoeff`, `nthmonomial`, `nthterm`, `poly`, `tcoeff`

Details:

- For a polynomial `p` of type `DOM_POLY` generated by `poly`, the function `F` must accept arguments from the coefficient ring of `p` and must produce corresponding results.
- A polynomial expression `f` is first converted to a polynomial with the variables given by `vars`. If no variables are given, they are searched for in `f`. See `poly` about details of the conversion. `FAIL` is returned if `f` cannot be converted to a polynomial. After applying the function `F`, the result is converted to an expression.
- `mapcoeffs` evaluates its arguments. Note, however, that polynomials of type `DOM_POLY` do not evaluate their coefficients for efficiency reasons. Cf. example 4.
- `mapcoeffs` is a function of the system kernel.

Example 1. The function `sin` is mapped to the coefficients of a polynomial expression in the indeterminates `x` and `y`:

```
>> mapcoeffs(3*x^3 + x^2*y^2 + 2, sin)

                        3            2  2
             sin(2) + x  sin(3) + x  y  sin(1)
```

The following call makes `mapcoeffs` regard this expression as a polynomial in `x`. Consequently, `y` is regarded as a parameter that becomes part of the coefficients:

```
>> mapcoeffs(3*x^3 + x^2*y^2 + 2, [x], sin)
```

```
                    3            2       2
sin(2) + x  sin(3) + x  sin(y )
```

The system function `_plus` adds its arguments. In the following call, it is used to add 2 to all coefficients by providing this shift as an additional argument:

```
>> mapcoeffs(c1*x^3 + c2*x^2*y^2 + c3, [x, y], _plus, 2)
```

```
               3               2   2
     c3 + x  (c1 + 2) + x  y  (c2 + 2) + 2
```

Example 2. The function `sin` is mapped to the coefficients of a polynomial in the indeterminates `x` and `y`:

```
>> mapcoeffs(poly(3*x^3 + x^2*y^2 + 2, [x, y]), sin)
```

```
                 3            2   2
poly(sin(3) x  + sin(1) x  y  + sin(2), [x, y])
```

In the following call, the polynomial has the indeterminate `x`. Consequently, `y` is regarded as a parameter that becomes part of the coefficients:

```
>> mapcoeffs(poly(3*x^3 + x^2*y^2 + 2, [x]), sin)
```

```
                 3        2   2
poly(sin(3) x  + sin(y ) x  + sin(2), [x])
```

A user-defined function is mapped to a polynomial:

```
>> F := (c, a1, a2) -> exp(c + a1 + a2):
   mapcoeffs(poly(x^3 + c*x, [x]), F, a1, a2)
```

```
                         3
poly(exp(a1 + a2 + 1) x  + exp(c + a1 + a2) x, [x])
```

```
>> delete F:
```

Example 3. We consider a polynomial over the integers modulo 7:

```
>> p := poly(x^3 + 2*x*y, [x, y], Dom::IntegerMod(7)):
```

A function to be applied to the coefficients must produce values in the coefficient ring of the polynomial:

```
>> mapcoeffs(p, c -> c^2)
```

```
      3
poly(x  + 4 x y, [x, y], Dom::IntegerMod(7))
```

The following call maps a function which converts its argument to an integer modulo 3. Such a return value is not a valid coefficient of `p`:

```
>> mapcoeffs(p, c -> Dom::IntegerMod(3)(expr(c)))

                              FAIL

>> delete p:
```

Example 4. Note that polynomials of type `DOM_POLY` do not evaluate their arguments:

```
>> delete a, x: p := poly(a*x, [x]): a := PI: p

                         poly(a x, [x])
```

Evaluation can be enforced by the function `eval`:

```
>> mapcoeffs(p, eval)

                         poly(PI x, [x])
```

We map the sine function to the coefficients of `p`. The polynomial does not evaluate its coefficient `sin(a)` to `0`:

```
>> mapcoeffs(p, sin)

                       poly(sin(a) x, [x])
```

The composition of `sin` and `eval` is mapped to the coefficients of the polynomial:

```
>> mapcoeffs(p, eval@sin)

                          poly(0, [x])

>> delete p, a:
```

`maprat` – apply a function to the "rationalization" of an expression

`maprat(object, f)` applies the function `f` to the "rationalized" object.

Call(s):

- `maprat(object, f <, inspect <, stop>>)`

Parameters:

`object`	—	an arithmetical expression, or a sequence, or a set, or a list of such expressions
`f`	—	a procedure or a functional expression
`inspect, stop`	—	sets of types or procedures

Return Value: an object returned by the function `f`.

Related Functions: `map`, `rationalize`

Details:

- `maprat(object, f, inspect, stop)` calls `rationalize(object, inspect, stop)` to generate a rational expression in some "temporary variables". This rationalized expression is used as input to the function `f`. Finally, in the return value of `f`, the "temporary variables" introduced by `rationalize` are replaced by the original subexpressions in `object`.
- See the help page of `rationalize` for details and default values of the parameters `inspect` and `stop`.

Example 1. The function `partfrac` computes a partial fraction decomposition of rational expressions. It cannot be applied to general expressions:

```
>> object := cos(x)/(cos(x)^2 - sin(x)^2): partfrac(object, x)

 Error: not a rational function [partfrac]
```

One may rationalize this expression to be able to apply `partfrac`:

```
>> rat := rationalize(object)

                 D1
           ---------, {D1 = cos(x), D2 = sin(x)}
             2     2
           D1  - D2
```

We compute the partial fraction decomposition of this rationalized expression and, finally, re-substitute the "temporary variables" `D1`, `D2`:

```
>> part := partfrac(op(rat, 1), D1)

                    1             1
               ----------- + -----------
               2 (D1 + D2)   2 (D1 - D2)

>> subs(part, op(rat, 2))
```

```
          1                    1
------------------- + -------------------
2 (cos(x) + sin(x))   2 (cos(x) - sin(x))
```

`maprat` provides a shortcut. We define a function `f` that computes the partial fraction decomposition of its argument with respect to the first indeterminate found by `indets`:

```
>> f := object -> partfrac(object, indets(object)[1]):
```

`maprat` applies this function after internal rationalization:

```
>> maprat(object, f)
```

```
          1                    1
------------------- + -------------------
2 (cos(x) + sin(x))   2 (cos(x) - sin(x))
```

```
>> delete object, rat, part, f:
```

Example 2. We apply the function `gcd` to two rationalized expressions. The first argument to `maprat` is a sequence of the two expressions `p`, `q`, which `gcd` takes as two parameters. Note the brackets around the sequence `p, q`:

```
>> p := (x - sqrt(2))*(x^2 + sqrt(3)*x - 1):
   q := (x - sqrt(2))*(x - sqrt(3)):
   maprat((p, q), gcd)
```

```
 1/2
2    - x
```

```
>> delete p, q:
```

`max` – the maximum of numbers

`max(x1, x2, ...)` returns the maximum of the numbers $x_1, x_2, \ldots$.

Call(s):

- `max(x1, x2, ...)`

Parameters:

`x1, x2, ...` — arbitrary MuPAD objects

Return Value: one of the arguments, a floating point number, or a symbolic `max` call.

Overloadable by: `x1, x2, ...`

Related Functions: `_leequal`, `_less`, `min`, `sysorder`

Details:

- If the arguments of `max` are either integers, rational numbers, or floating point numbers, then `max` returns the numerical maximum of these arguments.
- Arguments of type `DOM_INTERVAL` (floating point intervals) are replaced by their upper bounds.
- The call `max()` is illegal and leads to an error message. If there is only one argument `x1`, then `max` evaluates `x1` and returns it, after replacing floating point intervals by their upper bounds (see example 2).
- If one of the arguments is `infinity`, then `max` returns `infinity`. If an argument is `-infinity`, then it is removed from the argument list (see example 3).
- `max` returns an error when one of its arguments is a complex number or a floating point interval with non-zero imaginary part (see example 2).
- If one of the arguments is not a number, then a symbolic `max` call with the maximum of the numerical arguments and the remaining evaluated arguments is returned (see example 1).

  Nested `max` calls with symbolic arguments are rewritten as a single `max` call, i.e., they are flattened; see example 4.
- `max` does *not* react to properties of identifiers set via `assume`. Use `simplify` to handle this (see example 4).
- `max` is a function of the system kernel.

Example 1. `max` computes the maximum of integers, rational numbers, and floating point values:

```
>> max(-3/2, 7, 1.4)

                                   7
```

Floating point intervals are replaced by their upper limits:

```
>> max(2...3 union 6...7, 4)
```

```
7.0
```

If the argument list contains symbolic expressions, then a symbolic `max` call is returned:

```
>> delete b: max(-4, b + 2, 1, 3)

                    max(b + 2, 3)

>> max(sqrt(2), 1)

                         1/2
                    max(2   , 1)
```

Use `simplify` to simplify `max` expressions with constant symbolic arguments:

```
>> simplify(%)

                         1/2
                        2
```

Example 2. `max` with one argument returns the evaluated argument:

```
>> delete a: max(a), max(sin(2*PI)), max(2)

                    a, 0, 2
```

Even in this case, a floating point interval is replaced by its upper limit:

```
>> max(-10 ... 10)

                    10.0
```

Complex numbers lead to an error message:

```
>> max(0, 1, I)

 Error: Illegal argument [max]
```

Example 3. `infinity` is always the maximum of arbitrary arguments:

```
>> delete x: max(100000000000, infinity, x)

                    infinity
```

`-infinity` is removed from the argument list:

```
>> max(100000000000, -infinity, x)

                max(x, 100000000000)
```

Example 4. `max` does not take into account properties of identifiers set via `assume`:

```
>> delete a, b, c:
   assume(a > 0): assume(b > a, _and): assume(c > b, _and):
   max(a, max(b, c), 0)
```

```
                    max(a, b, c, 0)
```

An application of `simplify` yields the desired result:

```
>> simplify(%)
```

```
                          c
```

Changes:

- Arguments of type `DOM_INTERVAL` are handled.

`min` – the minimum of numbers

`min(x1, x2, ...)` returns the minimum of the numbers $x_1, x_2, \ldots$.

Call(s):

- `min(x1, x2, ...)`

Parameters:

`x1, x2, ...` — arbitrary MuPAD objects

Return Value: one of the arguments, a float, or a symbolic `min` call.

Overloadable by: `x1, x2, ...`

Related Functions: `_leequal`, `_less`, `min`, `sysorder`

Details:

- If the arguments of `min` are integers, rational numbers, or floating point numbers, then `min` returns the numerical minimum of these arguments.
- Arguments of type `DOM_INTERVAL` (floating point intervals) are replaced by their lower bounds.

- The call `min()` is illegal and leads to an error message. If there is only one argument `x1`, then `min` evaluates `x1` and returns it, unless it is a floating point interval, in which case the lower bound is returned. See example 2.
- If one of the arguments is `-infinity`, then `min` returns `-infinity`. If an argument is `infinity`, then it is removed from the argument list (see example 3).
- `min` returns an error when one of its arguments is a complex number or a floating point interval with on-zero imaginary part (see example 2).
- If one of the arguments is not a number, then a symbolic `min` call with the minimum of the numerical arguments and the remaining evaluated arguments is returned (see example 1).

  Nested `min` calls with symbolic arguments are rewritten as a single `min` call, i.e., they are flattened; see example 4.
- `min` does *not* react to properties of identifiers set via `assume`. Use `simplify` to handle this (see example 4).
- `min` is a function of the system kernel.

Example 1. `min` computes the minimum of integers, rational numbers, and floating point values:

```
>> min(-3/2, 7, 1.4)

                                -3/2
```

If the argument list contains symbolic expressions, then a symbolic `min` call is returned:

```
>> delete b: min(-4, b + 2, 1, 3)

                            min(b + 2, -4)

>> min(sqrt(2), 1)

                                 1/2
                             min(2   , 1)
```

Use `simplify` to simplify `min` expressions with constant symbolic arguments:

```
>> simplify(%)

                                  1
```

Floating point intervals are replaced by their lower borders:

```
>> min(2...3 union 6...7, 4)

                                 2.0
```

Example 2. `min` with one argument returns the evaluated argument:

```
>> delete a: min(a), min(sin(2*PI)), min(2)

                    a, 0, 2
```

Even in this situation, floating point intervals are replaced by their lower bounds:

```
>> min(-10...10)

                    -10.0
```

Complex numbers lead to an error message:

```
>> min(0, 1, I)

 Error: Illegal argument [min]
```

Example 3. `-infinity` is always the minimum of arbitrary arguments:

```
>> delete x: min(-100000000000, -infinity, x)

                    -infinity
```

`infinity` is removed from the argument list:

```
>> min(-100000000000, infinity, x)

               min(x, -100000000000)
```

Example 4. `min` does not take into account properties of identifiers set via `assume`:

```
>> delete a, b, c:
   assume(a > 0): assume(b > a, _and): assume(c > b, _and):
   min(a, min(b, c), 0)

                  min(a, b, c, 0)
```

An application of `simplify` yields the desired result:

```
>> simplify(%)

                    0
```

Changes:

- Arguments of type `DOM_INTERVAL` are handled.

`mod, modp, mods` – the modulo functions

`modp(x, m)` computes the unique nonnegative remainder on division of the integer x by the integer m.

`mods(x, m)` computes the integer r of least absolute value such that the integer $x - r$ is divisible by the integer m.

By default, `x mod m` and `_mod(x, m)` are both equivalent to `modp(x, m)`.

Call(s):

- `x mod m`
- `_mod(x, m)`
- `modp(x, m)`
- `mods(x, m)`

Parameters:

`x, m` — arithmetical expressions

Return Value: an arithmetical expression.

Overloadable by: `x, m`

Side Effects: By default the operator `mod` and the function `_mod` are equivalent to `modp`. This can be changed by assigning a new value to `_mod`; see example 5.

Related Functions: `/`, `div`, `divide`, `Dom::IntegerMod`, `frac`, `gcd`, `gcdex`, `igcd`, `igcdex`, `IntMod`, `powermod`

Details:

- If m is a nonzero integer and x is an integer, then both `modp` and `mods` return an integer r such that $x = qm + r$ holds for some integer q. In addition, we have $0 \leq r < |m|$ for `modp` and $-|m|/2 < r \leq |m|/2$ for `mods`. See example 2. These conditions uniquely define r in both cases. In the `modp` case, we have `q = x div m`.

- If m is a nonzero integer and x is a rational number, say $x = u/v$ for two nonzero coprime integers u and v, then `modp` and `mods` both compute an integral solution r of the congruence $vr \equiv u \bmod m$. To this end, they first compute an inverse w of v modulo m, such that $vw - 1$ is divisible by m. This only works if v is coprime to m, i.e., if their greatest common divisor is 1. Then `modp(u*w, m)` or `mods(u*w, m)`, respectively, as described above, is returned. Otherwise, if v and m are not coprime, then an error message is returned. See example 2.

  The number `x - modp(x, m)` is not an integral multiple of m in this case.

- If m is a (nonzero) rational number and x is an integer or a rational number, then both `modp` and `mods` return an integer or a rational number r such that $x = qm + r$ holds for some integer q. In addition, we have $0 \le r < |m|$ for `modp` and $-|m|/2 < r \le |m|/2$ for `mods`, and these conditions uniquely define r in both cases. See example 3.

- If the second argument m is 0, then an error message is returned.

- `_mod(x, m)` is the functional equivalent of the operator notation `x mod m`. See example 1.

- By default, `_mod` is equivalent to `modp`.

- The functions `modp` and `mods` can be used to redefine the modulo operator. E.g., after the assignment `_mod:=mods`, both the operator `mod` and the equivalent function `_mod` return remainders of least absolute value. See example 5.

- All functions return an error when one of the arguments is a floating point number, a complex number, or not an arithmetical expression.

- If one of the arguments is not a number, then a symbolic function call is returned. See example 4.

- `_mod`, `modp`, and `mods` are kernel functions.

Example 1. The example demonstrates the correspondence between the function `_mod` and the operator `mod`:

```
>> hold(_mod(23,5))

                              23 mod 5

>> 23 mod 5 = _mod(23,5)

                                3 = 3
```

Example 2. Here are some examples where the modulus is an integer. We see that `mod` and `modp` are equivalent by default:

```
>> 27 mod 3, 27 mod 4, modp(27, 4), mods(27, 4)
```

```
0, 3, 3, -1
```

```
>> 27 = (27 div 4)*4 + modp(27, 4)
```

```
27 = 27
```

Let us now compute 22/3 modulo 5. The greatest common divisor of 3 and 5 is 1, and 2 is an inverse of 3 modulo 5. Thus 22/3 modulo 5 equals $22 \cdot 2$ modulo 5:

```
>> modp(22/3, 5) = modp(22*2, 5),
   mods(22/3, 5) = mods(22*2, 5)
```

```
4 = 4, -1 = -1
```

The greatest common divisor of 15 and 27 is 3, so that 15 has no inverse modulo 27 and the following command fails:

```
>> modp(-22/15, 27)
```

```
 Error: Modular inverse does not exist
```

However, we can compute $-22/15$ modulo 26, since 15 and 26 are coprime:

```
>> -22/15 mod 26
```

```
2
```

Example 3. Here are some examples where the modulus is a rational number. We have $23/3 = 9 \cdot 4/5 + 7/15 = 10 \cdot 4/5 - 1/3$ and $23 = 28 \cdot 4/5 + 3/5 = 29 \cdot 4/5 - 1/5$. Thus we obtain:

```
>> modp(23/3, 4/5), mods(23/3, 4/5),
   modp(23, 4/5), mods(23, 4/5)
```

```
7/15, -1/3, 3/5, -1/5
```

Example 4. If one of the arguments is not a number, then a symbolic function call is returned:

```
>> delete x, m:
   x mod m, x mod 2, 2 mod m

                  x mod m, x mod 2, 2 mod m
```

`modp` and `mods` with non-numeric arguments are printed in the operator notation:

```
>> modp(x, m), mods(x, m)

                      x mod m, x mod m
```

Example 5. By default the binary operator `mod` and the equivalent function `_mod` are both equivalent to `modp`. This can be changed by redefining `_mod`:

```
>> 11 mod 7, modp(11,7), mods(11,7)

                          4, 4, -3
```

```
>> _mod := mods: 11 mod 7;
   _mod := modp:

                             -3
```

`multcoeffs` – multiply the coefficients of a polynomial with a factor

`multcoeffs(p, c)` multiplies all coefficients of the polynomial `p` with the factor `c`.

Call(s):

- `multcoeffs(p, c)`
- `multcoeffs(f, <vars,> c)`

Parameters:

`p`	—	a polynomial of type `DOM_POLY`
`c`	—	an arithmetical expression or an element of the coefficient ring of `p`
`f`	—	a polynomial expression
`vars`	—	a list of indeterminates of the polynomial: typically, identifiers or indexed identifiers

Return Value: a polynomial of type `DOM_POLY`, or a polynomial expression, or `FAIL`.

Overloadable by: `p, f`

Related Functions: `coeff`, `degree`, `degreevec`, `lcoeff`, `ldegree`, `lterm`, `nterms`, `nthcoeff`, `nthmonomial`, `nthterm`, `poly`, `tcoeff`

Details:

- A polynomial expression `f` is first converted to a polynomial with the variables given by `vars`. If no variables are given, they are searched for in `f`. See `poly` about details of the conversion. `FAIL` is returned if `f` cannot be converted to a polynomial. After multiplication with `c`, the result is converted to an expression.
- For a polynomial expression `f`, the factor `c` may be any arithmetical expression. For a polynomial `p` of type `DOM_POLY`, the factor `c` must be convertible to an element of the coefficient ring of `p`.
- `multcoeffs` is a function of the system kernel.

Example 1. Some simple examples:

```
>> multcoeffs(3*x^3 + x^2*y^2 + 2, 5)

                         3      2  2
                     15 x  + 5 x  y  + 10

>> multcoeffs(3*x^3 + x^2*y^2 + 2, c)

                              3      2  2
                    2 c + 3 c x  + c x  y

>> multcoeffs(poly(x^3 + 2, [x]), sin(y))

                             3
                poly(sin(y) x  + 2 sin(y), [x])
```

Example 2. Mathematically, `multcoeffs(f, c)` is the same as `f*c`. However, `multcoeffs` produces an expanded form of the product which depends on the indeterminates:

```
>> f := 3*x^3 + x^2*y^2 + 2:
   multcoeffs(f, [x], c), multcoeffs(f, [y], c),
   multcoeffs(f, [z], c)
```

```
          3      2  2      2  2          3
2 c + 3 c x  + c x  y , c x  y  + c (3 x  + 2),

         3    2  2
   c (3 x  + x  y  + 2)

>> delete f:
```

---

new – create a domain element

`new(T, object1, object2, ...)` creates a new element of the domain `T` with the internal representation `object1, object2, ...`.

Call(s):

- `new(T, object1, object2, ...)`

Parameters:

`T`	—	a MuPAD domain
`object1, object2, ...`	—	arbitrary MuPAD objects

Return Value: an element of the domain `T`.

Related Functions: `DOM_DOMAIN`, `domain`, `extop`, `extnops`, `extsubsop`, `newDomain`, `op`

Details:

- `new` is a low-level function for creating elements of library domains.

  The internal representation of a domain element comprises a reference to the corresponding domain and an arbitrary number of MuPAD objects, the internal operands of the domain element.

- `new(T, object1, object2, ...)` creates a new element of the domain `T`, whose internal representation is the sequence of operands `object1, object2, ...`, and returns this element.

  `new(T)` creates a new element of the domain `T`, whose internal representation is an empty sequence of operands.

- `new` is intended only for programmers implementing their own domains in MuPAD. You should never use `new` directly to generate elements of a predefined domain `T`; use the corresponding constructor `T(...)` instead, for the following reasons. The internal representation of the predefined MuPAD domains may be subject to changes more often than the interface provided by the constructor. Moreover, in contrast to `new`, the constructors usually perform argument checking. Thus using `new` directly may lead to invalid internal representations of MuPAD objects.

- New domains can be created via `newDomain`.

- You can access the operands of the internal representation of a domain element via `extop`, which, in contrast to `op`, cannot be overloaded for the domain. The function `op` is sometimes overloaded for a domain in order to hide the internal, technical representation of an object and to provide a more user friendly and intuitive interface.

- Similarly, the function `extnops` returns the number of operands of a domain element in the internal representation, and `extsubsop` modifies an operand in the internal representation. These functions, in contrast to the related functions `nops` and `subsop`, cannot be overloaded for a domain.

- You can write a constructor for your own domain `T` by providing a `"new"` method. This method is invoked whenever the user calls `T(arg1, arg2, ...)`. This is recommended since it provides a more elegant and intuitive user interface than `new`. The `"new"` method usually performs some argument checking and converts the arguments `arg1, arg2, ...` into the internal representation of the domain, using `new` (see example 1).

- `new` is a function of the system kernel.

Example 1. We create a new domain `Time` for representing clock times. The internal representation of an object of this domain has two operands: the hour and the minutes. Then we create a new domain element for the time 12 : 45:

```
>> Time := newDomain("Time"):
   a := new(Time, 12, 45)

                         new(Time, 12, 45)
```

The domain type of `a` is `Time`, the number of operands is `2`, and the operands are `12` and `45`:

```
>> domtype(a), extnops(a)

                              Time, 2

>> extop(a)
```

```
12, 45
```

We now implement a "new" method for our new domain Time, permitting several input formats. It expects either two integers, the hour and the minutes, or only one integer that represents the minutes, or a rational number or a floating point number, implying that the integral part is the hour and the fractional part represents a fraction of an hour corresponding to the minutes, or no arguments, representing midnight. Additionally, the procedure checks that the arguments are of the correct type:

```
>> Time::new := proc(HR = 0, MN = 0)
     local m;
   begin
     if args(0) = 2 and domtype(HR) = DOM_INT
        and domtype(MN) = DOM_INT then
       m := HR*60 + MN
     elif args(0) = 1 and domtype(HR) = DOM_INT then
       m := HR
     elif args(0) = 1 and domtype(HR) = DOM_RAT then
       m := trunc(float(HR))*60 + frac(float(HR))*60
     elif args(0) = 1 and domtype(HR) = DOM_FLOAT then
       m := trunc(HR)*60 + frac(HR)*60
     elif args(0) = 0 then
       m := 0
     else
       error("wrong number or type of arguments")
     end_if;
     new(Time, trunc(m/60), trunc(m) mod 60)
   end_proc:
```

Now we can use this method to create new objects of the domain Time, either by calling Time::new directly, or, preferably, by using the equivalent but shorter call Time(...):

```
>> Time::new(12, 45), Time(12, 45), Time(12 + 3/4)

     new(Time, 12, 45), new(Time, 12, 45), new(Time, 12, 45)

>> Time(), Time(8.25), Time(1/2)

       new(Time, 0, 0), new(Time, 8, 15), new(Time, 0, 30)
```

In order to have a nicer output for objects of the domain Time, we also define a "print" method (see the help page for print):

```
>> Time::print := proc(TM)
   begin
     expr2text(extop(TM, 1)) . ":" .
     stringlib::format(expr2text(extop(TM, 2)), 2, Right, "0")
   end_proc:
```

```
>> Time::new(12, 45), Time(12, 45), Time(12 + 3/4)

                        12:45, 12:45, 12:45

>> Time(), Time(8.25), Time(1/2)

                         0:00, 8:15, 0:30
```

`newDomain` – create a new data type (domain)

`newDomain(k)` creates a new domain with key `k`.

`newDomain(k, T)` creates a copy of the domain `T` with new key `k`.

`newDomain(k, t)` creates a new domain with key `k` and slots from the table `t`.

Call(s):

- `newDomain(k)`
- `newDomain(k, T)`
- `newDomain(k, t)`

Parameters:

`k` — an arbitrary object; typically a string
`T` — a domain
`t` — the slots of the domain: a table

Return Value: an object of type `DOM_DOMAIN`.

Further Documentation: The document “Axioms, Categories and Domains” is a detailed technical reference for domains.

Related Functions: `DOM_DOMAIN`, `domain`, `domtype`, `new`, `slot`

Details:

- Data types in MuPAD are called *domains*. `newDomain` is a low-level function for defining new data types. Cf. the corresponding entry in the Glossary for links to documentation about domains and more comfortable ways of defining new data types. The help page of `DOM_DOMAIN` contains a tutorial example for defining a new domain via `newDomain`.

- Technically, a domain is something like a table. The entries of this table are called *slots* or *methods*. They serve for extending the functionality of standard MuPAD functions, such as the arithmetic operations `+` and `*`, the special mathematical functions `exp` and `sin`, or the symbolic manipulation functions `simplify` and `normal`, to objects of a domain in a modular, object-oriented way, without the need to modify the source code of the standard function. This is known as *overloading*.

  The function `slot` and the equivalent operator `::` serve for defining and accessing a specific slot of a domain. The function `op` returns all slots of a domain.

- Each domain has a distinguished slot `"key"`, which is its unique identification. There can be no two different domains with the same key. Typically, but not necessarily, the key is a string. However, the key serves mainly for internal and output purposes. Usually a domain is assigned to an identifier immediately after its creation, and you access the domain via this identifier.

- If a domain with the given key already exists, `newDomain(k)` returns that domain; both other forms of calling `newDomain` yield an error.

- `newDomain` is a function of the system kernel.

Example 1. We create new domain with key `"my-domain"`. This key is also used for output, but without quotes:

```
>> T := newDomain("my-domain")

                                my-domain
```

You can create elements of this domain with the function `new`:

```
>> e := new(T, 42);
   domtype(e)

                          new(my-domain, 42)

                                my-domain
```

With the slot operator `::`, you can define a new slot or access an existing one:

```
>> op(T)

                          "key" = "my-domain"

>> T::key, T::myslot

                           "my-domain", FAIL
```

```
>> T::myslot := 42: op(T)

               "myslot" = 42, "key" = "my-domain"

>> T::myslot^2

                              1764
```

If a domain with key `k` already exists, then `newDomain(k)` does not create a new domain, but returns the existing domain instead:

```
>> T1 := newDomain("my-domain"):
   op(T1)

               "myslot" = 42, "key" = "my-domain"
```

Note that you cannot delete a domain; the command `delete T` only deletes the value of the identifier `T`, but does not destroy the domain with the key `"my-domain"`:

```
>> delete T, T1:
   T2 := newDomain("my-domain"):
   op(T2);
   delete T2:

               "myslot" = 42, "key" = "my-domain"
```

Example 2. There cannot exist different domains with the same key at the same time. Defining a slot for a domain implicitly changes all identifiers that have this domain as their value:

```
>> T := newDomain("1st"): T1 := T:
   op(T);
   op(T1);

                          "key" = "1st"

                          "key" = "1st"

>> T1::mySlot := 42:
   op(T);
   op(T1);

                  "mySlot" = 42, "key" = "1st"

                  "mySlot" = 42, "key" = "1st"
```

To avoid this, you can create a copy of a domain. You must reserve a new, unused key for that copy:

```
>> T2 := newDomain("2nd", T):
   T2::anotherSlot := infinity:
   op(T);
   op(T2);
```

```
                "mySlot" = 42, "key" = "1st"

  "anotherSlot" = infinity, "mySlot" = 42, "key" = "2nd"
```

```
>> delete T, T1, T2:
```

Example 3. You can provide a domain with slots already when creating it:

```
>> T := newDomain("3rd",
     table("myslot" = 42, "anotherSlot" = infinity)):
   op(T);
   T::myslot, T::anotherSlot
```

```
  "key" = "3rd", "anotherSlot" = infinity, "myslot" = 42

                       42, infinity
```

```
>> delete T:
```

---

next – skip a step in a loop

`next` interrupts the current step in `for`, `repeat`, and `while` loops. Execution proceeds with the next step of the loop.

Call(s):

- next
- `_next()`

Related Functions: `break`, `case`, `for`, `quit`, `repeat`, `return`, `while`

Details:

- The `next` statement is equivalent to the function call `_next()`. The return value is the void object of type `DOM_NULL`.

- Inside `for`, `repeat`, and `while` loops, the `next` statement interrupts the current step of the loop. In `for` statements, the loop variable is incremented and execution continues at the beginning of the loop. Similarly, the control conditions at the beginning of a `while` loop and in the `until` clause of a `repeat` loop are verified, before execution continues at the beginning of the loop.
- Outside `for`, `repeat`, and `while` loops, the `next` statement has no effect.
- `_next` is a function of the system kernel.

Example 1. In the following `for` loop, any step with even `i` is skipped:

```
>> for i from 1 to 5 do
     if testtype(i, Type::Even) then next end_if;
     print(i)
   end_for:
```

```
1

3

5
```

In the following `repeat` loop, all steps with odd `i` are skipped:

```
>> i := 0:
   repeat
     i := i + 1;
     if testtype(i, Type::Odd) then next end_if;
     print(i)
   until i >= 5 end_repeat:
```

```
2

4
```

```
>> delete i:
```

`nextprime` – the next prime number

`nextprime(m)` returns the smallest prime number larger than or equal to `m`.

Call(s):

- `nextprime(m)`

Parameters:

`m` — an arithmetical expression

Return Value: a prime number or a symbolic call to `nextprime`.

Related Functions: `ifactor`, `igcd`, `ilcm`, `isprime`, `ithprime`, `numlib::prevprime`

Details:

- If the argument `m` is an integer, then `nextprime` returns the smallest prime number larger than or equal to `m`. A symbolic call of type `"nextprime"` is returned, if the argument is not of type `Type::Numeric`. An error occurs if the argument is a number that is not an integer.
- The first prime number is 2.
- `nextprime` is a function of the system kernel.

Example 1. The first prime number is computed:

```
>> nextprime(-13)

                              2
```

If the argument of `nextprime` is a prime number, this number is returned:

```
>> nextprime(11)

                              11
```

We compute a large prime:

```
>> nextprime(56475767478567)

                        56475767478601
```

Symbolic arguments lead to a symbolic call:

```
>> nextprime(x)

                         nextprime(x)
```

Background:

- nextprime uses a fast probabilistic prime number test (Miller-Rabin test) to decide if the computed result is a prime number. The result returned by nextprime is either a prime number or a strong pseudo-prime for 10 randomly chosen bases.
- Reference: Michael O. Rabin, Probabilistic algorithms, in J. F. Traub, ed., *Algorithms and Complexity*, Academic Press, New York, 1976, pp. 21-39.

---

nops – the number of operands

`nops(object)` returns the number of operands of the object.

Call(s):

- `nops(object)`

Parameters:

`object` — an arbitrary MuPAD object

Return Value: a nonnegative integer.

Overloadable by: `object`

Related Functions: `extnops`, `extop`, `extsubsop`, `length`, `op`, `subsop`

Details:

- See the help page of `op` for details on MuPAD's concept of "operands".
- For sets, lists, and tables, the function `nops` returns the number of elements or entries, respectively. Note that expressions of type `DOM_EXPR` and arrays have a 0-th operand which is *not counted* by `nops`. For arrays, also non-initialized elements are counted by `nops`.
- The void object `null()` of type `DOM_NULL`, the empty list `[]`, the empty set `{}`, and the empty table `table()` have no operands: `nops` returns `0`. Cf. example 1.
- Integers of domain type `DOM_INT`, real floating point numbers of domain type `DOM_FLOAT`, Boolean constants of domain type `DOM_BOOL`, identifiers of domain type `DOM_IDENT`, and strings of domain type `DOM_STRING` are 'atomic' objects having only 1 operand: the object itself. Rational numbers of domain type `DOM_RAT` and complex numbers of domain type

DOM_COMPLEX have 2 operands: the numerator and denominator and the real part and imaginary part, respectively. Cf. example 2.

- In contrast to most other MuPAD functions, nops does not flatten expression sequences. Cf. example 3.

- nops is a function of the system kernel.

Example 1. The following expression has the type "_plus" and the three operands a*b, 3*c, and d:

```
>> nops(a*b + 3*c + d)

                                   3
```

For sets and lists, nops returns the number of elements. Note that the sublist [1, 2, 3] and the subset {1, 2} each count as one operand in the following examples:

```
>> nops({a, 1, [1, 2, 3], {1, 2}})

                                   4

>> nops([[1, 2, 3], 4, 5, {1, 2}])

                                   4
```

Empty objects have no operands:

```
>> nops(null()), nops([ ]), nops({}), nops(table())

                               0, 0, 0, 0
```

The number of operands of a symbolic function call is the number of arguments:

```
>> nops(f(3*x, 4, y + 2)), nops(f())

                                  3, 0
```

Example 2. Integers and real floating point numbers only have one operand:

```
>> nops(12), nops(1.41)

                                  1, 1
```

The same holds true for strings; use length to query the length of a string:

```
>> nops("MuPAD"), length("MuPAD")
```

```
1, 5
```

The number of operands of a rational number or a complex number is 2, even if the real part is zero:

```
>> nops(-3/2), nops(1 + I), nops(2*I)

                        2, 2, 2
```

A function environment has 3 and a procedure has 13 operands:

```
>> nops(sin), nops(op(sin, 1))

                         3, 13
```

Example 3. Expression sequences are not flattened by `nops`:

```
>> nops((1, 2, 3))

                           3
```

In contrast to the previous call, the following command calls `nops` with three arguments:

```
>> nops(1, 2, 3)

 Error: Wrong number of arguments [nops]
```

`norm` – compute the norm of a matrix, a vector, or a polynomial

`norm(M, kM)` computes the norm of index `kM` of the matrix `M`.

`norm(v, kv)` computes the norm of index `kv` of the vector `v`.

`norm(p, kp)` computes the norm of index `kp` of the polynomial `p`.

Call(s):

- `norm(M <, kM>)`
- `norm(v <, kv>)`
- `norm(p <, kp>)`
- `norm(f <, vars> <, kp>)`

Parameters:

`M`	—	a matrix of domain type `Dom::Matrix(...)`
`kM`	—	the index of the matrix norm: either 1, or *Frobenius* or *Infinity*. The default value is *Infinity*.
`v`	—	a vector (a 1-dimensional matrix)
`kv`	—	the index of the vector norm: either a positive integer, or *Frobenius*, or *Infinity*. The default value is *Infinity*.
`p`	—	a polynomial generated by `poly`
`f`	—	a polynomial expression
`vars`	—	a list of identifiers or indexed identifiers, interpreted as the indeterminates of `f`
`kp`	—	the index of the norm of the polynomial: a real number ≥ 1. If no index is specified, the maximum norm (of index infinity) is computed.

Return Value: an arithmetical expression.

Overloadable by: `p, f`

Related Functions: `coeff`, `float`, `matrix`, `poly`

Details:

- In MuPAD, there is no difference between matrices and vectors: a vector is a matrix of dimension $1 \times n$ or $n \times 1$, respectively.
- For an $m \times n$ matrix `M` $= (M_{ij})$ with $\min(m, n) > 1$, only the 1-norm (maximum column sum)

  $$\texttt{norm}(\texttt{M}, 1) = \max_{j=1,\ldots,n} \sum_{i=1}^{m} |M_{ij}|,$$

  the Frobenius norm

  $$\texttt{norm}(\texttt{M}, \texttt{Frobenius}) = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} |M_{ij}|^2},$$

  and the ∞-norm (maximum row sum)

  $$\texttt{norm}(\texttt{M}) = \texttt{norm}(\texttt{M}, \textit{Infinity}) = \max_{i=1,\ldots,m} \sum_{j=1}^{n} |M_{ij}|$$

  can be computed. The 1-norm and the *Infinity*-norm are operator norms with respect to the corresponding norms on the vector spaces the matrix is acting upon.

  For numerical matrices, the spectral norm (the operator norm with respect to the Euclidean norm (index 2)) is the largest singular value. It can be computed via `numeric::singularvalues`.

- For vectors $\mathtt{v} = (v_i)$, represented by matrices of dimension $1 \times n$ or $n \times 1$, norms with arbitrary positive integer indices k as well as *Infinity* can be computed. For integers $k > 1$, the vector norms are given by

$$\mathtt{norm}(\mathtt{v}, \mathtt{k}) = \Big(\sum_{i=1}^{n} |v_i|^k \Big)^{1/k}$$

for column vectors as well as for row vectors.

For indices 1, *Infinity*, and *Frobenius*, the vector norms are given by the corresponding matrix norms. For column vectors, the 1-norm is the sum norm

$$\mathtt{norm}(\mathtt{v}, 1) = \sum_{i=1}^{n} |v_i|,$$

the *Infinity*-norm is the maximum norm

$$\mathtt{norm}(\mathtt{v}) = \mathtt{norm}(\mathtt{v}, \mathtt{Infinity}) = \max(|v_1|, \ldots, |v_n|)$$

(this is the limit of the k-norms as k tends to infinity).

For row vectors, the 1-norm is the maximum norm, whilst the *Infinity*-norm is the sum norm.

The Frobenius norm coincides with `norm(v, 2)` for both column and row vectors.

Cf. example 2.

- Matrices and vectors may contain symbolic entries. No internal float conversion is applied.

- For matrix and vector norms, also refer to the help page of `Dom::Matrix` (note that the function `matrix` generates matrices of type `Dom::Matrix()`).

- For polynomials `p` with coefficients c_i, the norms are given by

$$\mathtt{norm}(\mathtt{p}) = \max |c_i| \ , \quad \mathtt{norm}(\mathtt{p}, \mathtt{k}) = \Big(\sum_i |c_i|^k \Big)^{1/k}.$$

Also multivariate polynomials are accepted by `norm`. The coefficients with respect to all indeterminates are taken into account.

- For polynomials, only numerical norms can be computed. The coefficients of the polynomial must not contain symbolic parameters that cannot be converted to floating point numbers. Coefficients containing symbolic numerical expressions such as `PI+1`, `sqrt(2)` etc. are accepted. Internally, they are converted to floating point numbers. Cf. example 3.

- For indices `k> 1`, `norm(p, k)` always returns a floating point number. The 1-norm produces an exact result if all coefficients are integers or rational numbers. The ∞-norm `norm(p)` produces an exact result, if the coefficient of largest magnitude is an integer or a rational number. In all other cases, also the 1-norm and the ∞-norm produce floating point numbers. Cf. example 3.

- For polynomials over the coefficient ring IntMod(m), `norm` produces an error.
- If the coefficient ring of the polynomial is a domain, it must implement the method `"norm"`. This method must return the norm of the coefficients as a number or as a numerical expression that can be converted to a floating point number via `float`. With the coefficient norms $\|c_i\|$, `norm(p)` computes the maximum norm $\max_i \|c_i\|$; `norm(p, k)` computes $(\sum_i \|c_i\|^k)^{1/k}$.
- A polynomial expression `f` is internally converted to the polynomial `poly(f)`. If a list of indeterminates is specified, the norm of the polynomial `poly(f, vars)` is computed.
- For polynomials and polynomial expressions, the norms are computed by a function of the system kernel.

Example 1. We compute various norms of a 2×3 matrix:

```
>> M := matrix([[2, 5, 8], [-2, 3, 5]]):
   norm(M) = norm(M, Infinity), norm(M, 1), norm(M, Frobenius)

                                          1/2
                        15 = 15, 13, 131
```

For matrices, `norm` produces exact symbolic results:

```
>> M := matrix([[2/3, 63, PI],[x, y, z]]): norm(M)

            max(PI + 191/3, abs(x) + abs(y) + abs(z))

>> norm(M, 1)

           max(abs(x) + 2/3, abs(y) + 63, PI + abs(z))

>> norm(M, Frobenius)

          2         2          2          2            1/2
      (PI  + abs(x)  + abs(y)   + abs(z)   + 35725/9)

>> delete M:
```

Example 2. A column vector `col` and a row vector `row` are considered:

```
>> col := matrix([x1, PI]): row := matrix([[x1, PI]]): col, row
```

```
                      +-    -+
                      |  x1  |  +-        -+
                      |      |, | x1, PI |
                      |  PI  |  +-        -+
                      +-    -+
```

```
>> norm(col, 2) = norm(row, 2)
```

```
                          2 1/2                          2 1/2
    (x1 conjugate(x1) + PI )    = (x1 conjugate(x1) + PI )
```

```
>> norm(col, 3) = norm(row, 3)
```

```
                3          3 1/3      3          3 1/3
            (PI  + abs(x1) )    = (PI  + abs(x1) )
```

Note that the norms of index 1 and *Infinity* have exchanged meanings for column and row vectors:

```
>> norm(col, 1) = norm(row, Infinity)

                  PI + abs(x1) = PI + abs(x1)

>> norm(col, Infinity) = norm(row, 1)

              max(abs(x1), PI) = max(abs(x1), PI)

>> delete col, row:
```

Example 3. The norms of some polynomials are computed:

```
>> p := poly(3*x^3 + 4*x, [x]): norm(p), norm(p, 1)

                              4, 7
```

If the coefficients are not integers or rational numbers, automatic conversion to floating point numbers occurs:

```
>> p := poly(3*x^3 + sqrt(2)*x + PI, [x]): norm(p), norm(p, 1)

                   3.141592654, 7.555806216
```

Floating point numbers are always produced for indices > 1:

```
>> p := poly(3*x^3 + 4*x + 1, [x]):
   norm(p, 1), norm(p, 2), norm(p, 5), norm(p, 10), norm(p)

          8, 5.099019514, 4.174686339, 4.021974513, 4

>> delete p:
```

Example 4. The norms of some polynomial expressions are computed:

```
>> norm(x^3 + 1, 1), norm(x^3 + 1, 2), norm(x^3 + PI)

                      2, 1.414213562, 1
```

The following call yields an error, because the expression is regarded as a polynomial in `x`. Consequently, symbolic coefficients $6\,y$ and $9\,y^2$ are found which are not accepted:

```
>> f := 6*x*y + 9*y^2 + 2: norm(f, [x])

 Error: Illegal argument [norm]
```

As a bivariate polynomial with the indeterminates `x` and `y`, the coefficients are 6, 9, and 2. Now, norms can be computed:

```
>> norm(f, [x, y], 1), norm(f, [x, y], 2), norm(f, [x, y])

                      17, 11.0, 9

>> delete f:
```

`normal` – normalize an expression

`normal(f)` returns the normal form of the rational expression `f`. This is a rational expression with expanded numerator and denominator whose greatest common divisor is 1.

`normal(object)` replaces the operands of `object` by their normalized form.

Call(s):

- `normal(f<, List>)`
- `normal(object)`

Parameters:

`f`	—	an arithmetical expression
`object`	—	a polynomial of type `DOM_POLY`, a list, a set, a table, an array, an equation, an inequality, or a range

Options:

`List`	—	return a list consisting of the numerator and denominator of `f`.

Return Value: an object of the same type as the input object. If the option *List* was given, a list of two arithmetical expressions.

Overloadable by: `object`

Further Documentation: Chapter "Manipulating Expressions" of the Tutorial.

Related Functions: `collect`, `combine`, `denom`, `expand`, `factor`, `gcd`, `indets`, `numer`, `partfrac`, `rationalize`, `rectform`, `rewrite`, `simplify`

Details:

- If the argument `f` contains non-rational subexpressions such as `sin(x)`, `x^(-1/3)` etc., then these are replaced by auxiliary variables before normalization. After normalization, these variables are replaced by the normalization of the original subexpressions. Algebraic dependencies of the subexpressions are not taken into account. The operands of the non-rational subexpressions are normalized recursively.
- If the argument `f` contains floating point numbers, then these are replaced by rational approximants (see `numeric::rationalize`). In the end, `float` is applied to the result.
- For special objects, `normal` is automatically mapped to its operands. In particular, if `object` is a polynomial of domain type `DOM_POLY`, then its coefficients are normalized. Further, if `object` is a set, a list, a table or an array, respectively, then `normal` is applied to all entries. Further, the left hand side and the right hand side of equations (type `"_equal"`), inequalities (type `"_unequal"`) and relations (type `"_less"` or `"_leequal"`) are normalized. Further, the operands of ranges (type `"_range"`) are normalized automatically.

Example 1. We compute the normal form of some rational expressions:

```
>> normal(x^2 - (x + 1)*(x - 1))

                                1

>> normal((x^2 - 1)/(x + 1))

                              x - 1

>> normal(1/(x + 1) + 1/(y - 1))

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

The following expression should be regarded as a rational expression in the "indeterminates" y and sin(x):

```
>> normal(1/sin(x)^2 + y/sin(x))

                          y sin(x) + 1
                          ------------
                                   2
                             sin(x)
```

Example 2. In the following, we give examples of non-rational expressions as argument. First, we normalize the entries of a list:

```
>> [(x^2 - 1)/(x + 1), x^2 - (x + 1)*(x - 1)]

               --  2                          --
               |  x  - 1   2                   |
               |  ------, x  - (x - 1) (x + 1) |
               --  x + 1                      --

>> normal(%)

                           [x - 1, 1]
```

The coefficients of polynomials are normalized:

```
>> poly((x^2-1)/(x+1)*Y^2 + (x^2-(x+1)*(x-1))*Y - 1, [Y])

       / /  2     \                                       \
       | | x  - 1 |  2     2                              |
   poly| | ------ | Y  + (x  - (x - 1) (x + 1)) Y - 1, [Y] |
       \ \  x + 1 /                                       /

>> normal(%)

                                2
                 poly((x - 1) Y  + Y - 1, [Y])
```

Example 3. If called with the option *List*, normal returns a list consisting of the numerator and the denominator of the input.

```
>> normal((x^2-1)/(x^2+2*x+1), List)

                         [x - 1, x + 1]
```

Note that normal(f, List) is *not* the same as [numer(f), denom(f)]:

```
>> [numer, denom]((x^2-1)/(x^2+2*x+1))

                       2              2
                     [x  - 1, 2 x + x  + 1]
```

Changes:

- Floating point numbers in the input are not longer replaced by auxiliary identifiers, but by rational approximants now.
- The new option *List* was introduced.

`nterms` – the number of terms of a polynomial

`nterms(p)` returns the number of terms of the polynomial `p`.

Call(s):

- `nterms(p)`
- `nterms(f <, vars>)`

Parameters:

`p`	—	a polynomial of type `DOM_POLY`
`f`	—	a polynomial expression
`vars`	—	a list of indeterminates of the polynomial: typically, identifiers or indexed identifiers

Return Value: a nonnegative number. `FAIL` is returned if the input cannot be converted to a polynomial.

Overloadable by: `p`

Related Functions: `coeff`, `degree`, `degreevec`, `ground`, `lcoeff`, `ldegree`, `lmonomial`, `lterm`, `nthcoeff`, `nthmonomial`, `nthterm`, `poly`, `poly2list`, `tcoeff`

Details:

- If the first argument `f` is not element of a polynomial domain, then `nterms` converts the expression to a polynomial via `poly(f)`. If a list of indeterminates is specified, then the polynomial `poly(f, vars)` is considered.
- A zero polynomial has no terms: the return value is 0.
- `nterms` is a function of the system kernel.

Example 1. We give some self explaining examples:

```
>> nterms(x^2*y^2 + x^2 + y + 2, [x, y])

                                4

>> nterms(poly(x^2*y^2 + x^2 + y + 2))

                                4

>> nterms(poly(0, [x]))

                                0
```

Example 2. The following polynomial expression may be regarded as a polynomial in different ways:

```
>> f := x^2*y^2 + x^2 + y + 2:
   nterms(f, [x]), nterms(f, [y]), nterms(f, [x, y]),
   nterms(f, [z])

                          2, 3, 4, 1

>> delete f:
```

`nthcoeff` – the n-th non-zero coefficient of a polynomial

`nthcoeff(p, n)` returns the `n`-th non-zero coefficient of the polynomial `p`.

Call(s):

- `nthcoeff(p, <vars,> n <, order>)`

Parameters:

`p`	—	a polynomial of type `DOM_POLY` or a polynomial expression
`vars`	—	a list of indeterminates of the polynomial: typically, identifiers or indexed identifiers
`n`	—	a positive integer
`order`	—	the term ordering: either *`LexOrder`* or *`DegreeOrder`* or *`DegInvLexOrder`* or a user-defined term ordering of type `Dom::MonomOrdering`. The default is the lexicographical ordering *`LexOrder`*.

Return Value: an element of the coefficient domain of the polynomial. An expression is returned if a polynomial expression is used as input. `FAIL` is returned if `n` is larger than the actual number of terms.

Overloadable by: `p`

Related Functions: `coeff`, `collect`, `degree`, `degreevec`, `ground`, `lcoeff`, `ldegree`, `lmonomial`, `lterm`, `nterms`, `nthmonomial`, `nthterm`, `poly`, `poly2list`, `tcoeff`

Details:

- The argument `p` can either be a polynomial expression, or a polynomial generated by `poly`, or an element of some polynomial domain overloading `nthcoeff`.
- If a list of indeterminates is provided, then `p` is regarded as a polynomial in these indeterminates. Note that the specified list does not have to coincide with the indeterminates of the input polynomial.
- The "first" coefficient is the leading coefficient as returned by `lcoeff`, the "last" coefficient is the trailing coefficient as returned by `tcoeff`.
- `nthcoeff` returned the `n`-th non-zero coefficient with respect to the lexicographical ordering, unless a different ordering is specified via the argument `order`. Cf. example 3.
- The result of `nthcoeff` is not fully evaluated. Evaluation can be enforced by the function `eval`. Cf. example 4.
- A zero polynomial has no terms: `nthcoeff` returns `FAIL`.
- `nthcoeff` is a library routine. If no term ordering is specified, the arguments are passed to a fast kernel routine.

Example 1. We give some self explaining examples:

```
>> p := poly(100*x^100 + 49*x^49 + 7*x^7, [x]):
   nthcoeff(p, 1), nthcoeff(p, 2), nthcoeff(p, 3)

                        100, 49, 7

>> nthcoeff(p, 4)

                           FAIL

>> nthcoeff(poly(0, [x]), 1)

                           FAIL

>> delete p:
```

Example 2. We demonstrate how the indeterminates influence the result:

```
>> p := 2*x^2*y + 3*x*y^2 + 6:
   nthcoeff(p, [x, y], 2), nthcoeff(p, [y, x], 2)

                              3, 2

>> p := poly(2*x^2*y + 3*x*y^2 + 6, [x, y]):
   nthcoeff(p, 1), nthcoeff(p, [y, x], 1)

                              2, 3

>> delete p:
```

Example 3. We demonstrate the effect of various term orders:

```
>> p := poly(5*x^4 + 4*x^3*y*z^2 + 3*x^2*y^3*z + 2, [x, y, z])

               4      3    2      2  3
        poly(5 x  + 4 x  y z  + 3 x  y  z + 2, [x, y, z])

>> nthcoeff(p, 1), nthcoeff(p, 1, DegreeOrder),
   nthcoeff(p, 1, DegInvLexOrder)

                             5, 4, 3
```

The following call uses the reverse lexicographical order on 3 indeterminates:

```
>> nthcoeff(p, 1, Dom::MonomOrdering(RevLex(3)))

                                3

>> delete p:
```

Example 4. We demonstrate the evaluation strategy of `nthcoeff`:

```
>> p := poly(3*x^3 + 6*x^2*y^2 + 2, [x]): y := 4:
   nthcoeff(p, 2)

                                 2
                              6 y
```

Evaluation is enforced by `eval`:

```
>> eval(%)

                               96
```

```
>> delete p, y:
```

---

nthmonomial – the *n*-th monomial of a polynomial

`nthmonomial(p, n)` returns the `n`-th non-trivial monomial of the polynomial `p`.

Call(s):

- `nthmonomial(p, <vars,> n <, order>)`

Parameters:

`p`	—	a polynomial of type `DOM_POLY` or a polynomial expression
`vars`	—	a list of indeterminates of the polynomial: typically, identifiers or indexed identifiers
`n`	—	a positive integer
`order`	—	the term ordering: *LexOrder*, or *DegreeOrder*, or *DegInvLexOrder*, or a user-defined term ordering of type `Dom::MonomOrdering`. The default is the lexicographical ordering *LexOrder*.

Return Value: a polynomial of the same type as `p`. An expression is returned if `p` is an expression. `FAIL` is returned if `n` is larger than the actual number of terms of the polynomial.

Overloadable by: `p`

Related Functions: `coeff`, `degree`, `degreevec`, `ground`, `lcoeff`, `ldegree`, `lmonomial`, `lterm`, `nterms`, `nthcoeff`, `nthterm`, `poly`, `poly2list`, `tcoeff`

Details:

- The argument `p` can either be a polynomial expression, or a polynomial generated by `poly`, or an element of some polynomial domain overloading `nthmonomial`.
- If a list of indeterminates is provided, then `p` is regarded as a polynomial in these indeterminates. The return value is a polynomial in these indeterminates as well. Note that the specified list does not have to coincide with the indeterminates of the input polynomial.
- The "first" monomial is the leading monomial as returned by `lmonomial`.

- nthmonomial returned the n-th non-trivial monomial with respect to the lexicographical ordering, unless a different ordering is specified via the argument order. Cf. example 3.
- The result of nthmonomial is not fully evaluated. It can be evaluated by the functions mapcoeffs and eval. Cf. example 5.
- A zero polynomial has no terms: nthmonomial returns FAIL.
- nthmonomial is a library routine. If no term ordering is specified, the arguments are passed to a fast kernel routine.

Example 1. We give some self explaining examples:

```
>> p := poly(100*x^100 + 49*x^49 + 7*x^7, [x]):
   nthmonomial(p, 1), nthmonomial(p, 2), nthmonomial(p, 3)

              100                  49                7
    poly(100 x    , [x]), poly(49 x   , [x]), poly(7 x , [x])

>> nthmonomial(p, 4)

                              FAIL

>> nthmonomial(poly(0, [x]), 1)

                              FAIL

>> delete p:
```

Example 2. We demonstrate how the indeterminates influence the result:

```
>> p := 2*x^2*y + 3*x*y^2 + 6:
   nthmonomial(p, [x, y], 2), nthmonomial(p, [y, x], 2)

                              2     2
                         3 x y , 2 x  y

>> p := poly(2*x^2*y + 3*x*y^2 + 6, [x, y]):
   nthmonomial(p, 2), nthmonomial(p, [y, x], 2)

                    2                     2
          poly(3 x y , [x, y]), poly(2 y x , [y, x])

>> delete p:
```

Example 3. We demonstrate the effect of various term orders:

```
>> p := poly(5*x^4 + 4*x^3*y*z^2 + 3*x^2*y^3*z + 2, [x, y, z]):
   nthmonomial(p, 1), nthmonomial(p, 1, DegreeOrder),
   nthmonomial(p, 1, DegInvLexOrder)

         4                        3    2
 poly(5 x , [x, y, z]), poly(4 x  y z , [x, y, z]),

            2  3
    poly(3 x  y  z, [x, y, z])

>> delete p:
```

Example 4. This example features a user defined term ordering. Here we use the reverse lexicographical order on 3 indeterminates:

```
>> order := Dom::MonomOrdering(RevLex(3)):
   p := poly(5*x^4 + 4*x^3*y*z^2 + 3*x^2*y^3*z + 2, [x, y, z]):
   nthmonomial(p, 2, order)

                        3    2
                 poly(4 x  y z , [x, y, z])
```

The following call produces all monomials:

```
>> nthmonomial(p, i, order) $ i = 1..nterms(p)

         2  3                          3    2
 poly(3 x  y  z, [x, y, z]), poly(4 x  y z , [x, y, z]),

            4
    poly(5 x , [x, y, z]), poly(2, [x, y, z])

>> delete order, p:
```

Example 5. We demonstrate the evaluation strategy of `nthmonomial`:

```
>> p := poly(3*x^3 + 6*x^2*y^2 + 2, [x]): y := 4:
   nthmonomial(p, 2)

                            2   2
                   poly((6 y ) x , [x])
```

Evaluation is enforced by `eval`:

```
>> mapcoeffs(%, eval)

                                   2
                        poly(96 x , [x])

>> delete p, y:
```

---

nthterm – the n-th term of a polynomial

nthterm(p, n) returns the n-th non-zero term of the polynomial p.

Call(s):

- nthterm(p, <vars,> n <, order>)

Parameters:

p	—	a polynomial of type DOM_POLY or a polynomial expression
vars	—	a list of indeterminates of the polynomial: typically, identifiers or indexed identifiers
n	—	a positive integer
order	—	the term ordering: either *LexOrder* or *DegreeOrder* or *DegInvLexOrder* or a user-defined term ordering of type Dom::MonomOrdering. The default is the lexicographical ordering *LexOrder*.

Return Value: a polynomial of the same type as p. An expression is returned if a polynomial expression is used as input. FAIL is returned if n is larger than the actual number of terms of the polynomial.

Overloadable by: p

Related Functions: coeff, degree, degreevec, ground, lcoeff, ldegree, lmonomial, lterm, nterms, nthcoeff, nthmonomial, poly, poly2list, tcoeff

---

Details:

- The argument p can either be a polynomial expression, or a polynomial generated by poly, or an element of some polynomial domain overloading nthterm.
- The identity nthterm(p, n) nthcoeff(p, n) = nthmonomial(p, n) holds.

- If a list of indeterminates is provided, then `p` is regarded as a polynomial in these indeterminates. The return value is a polynomial in these indeterminates as well. Note that the specified list does not have to coincide with the indeterminates of the input polynomial.
- The "first" term is the leading term as returned by `lterm`.
- `nthterm` returned the `n`-th non-zero term with respect to the lexicographical ordering, unless a different ordering is specified via the argument `order`. Cf. example 3.
- A zero polynomial has no terms: `nthterm` returns `FAIL`.
- `nthterm` is a library routine. If no term ordering is specified, the arguments are passed to a fast kernel routine.

Example 1. We give some self explaining examples:

```
>> p := poly(100*x^100 + 49*x^49 + 7*x^7, [x]):
   nthterm(p, 1), nthterm(p, 2), nthterm(p, 3)

                 100                49                7
         poly(x    , [x]), poly(x   , [x]), poly(x , [x])

>> nthterm(p, 4)

                               FAIL

>> nthterm(poly(0, [x]), 1)

                               FAIL

>> delete p:
```

Example 2. We demonstrate how the indeterminates influence the result:

```
>> p := 2*x^2*y + 3*x*y^2 + 6:
   nthterm(p, [x, y], 2), nthterm(p, [y, x], 2)

                               2   2
                            x y , x  y

>> p := poly(2*x^2*y + 3*x*y^2 + 6, [x, y]):
   nthterm(p, 2), nthterm(p, [y,x], 2)

                      2                    2
             poly(x y , [x, y]), poly(y x , [y, x])

>> delete p:
```

Example 3. We demonstrate the effect of various term orders:

```
>> p := poly(5*x^4 + 4*x^3*y*z^2 + 3*x^2*y^3*z + 2, [x,y,z]):
   nthterm(p, 1), nthterm(p, 1, DegreeOrder),
   nthterm(p, 1, DegInvLexOrder)

       4                    3    2
 poly(x , [x, y, z]), poly(x  y z , [x, y, z]),

          2  3
    poly(x  y  z, [x, y, z])

>> delete p:
```

Example 4. This example features a user defined term ordering. Here we use the reverse lexicographical order on 3 indeterminates:

```
>> order := Dom::MonomOrdering(RevLex(3)):
   p := poly(5*x^4 + 4*x^3*y*z^2 + 3*x^2*y^3*z + 2, [x,y,z]):
   nthterm(p, 2, order)

                          3    2
                    poly(x  y z , [x, y, z])
```

The following call produces all terms:

```
>> nthterm(p, i, order) $ i = 1..nterms(p)

       2  3                       3    2
 poly(x  y  z, [x, y, z]), poly(x  y z , [x, y, z]),

          4
    poly(x , [x, y, z]), poly(1, [x, y, z])

>> delete order, p:
```

Example 5. The n-th monomial is the product of the n-th coefficient and the n-th term:

```
>> p := poly(2*x^2*y + 3*x*y^2 + 6, [x, y]):
   mapcoeffs(nthterm(p, 2), nthcoeff(p, 2)) =
   nthmonomial(p, 2)

                   2                       2
          poly(3 x y , [x, y]) = poly(3 x y , [x, y])
```

```
>> delete p:
```

---

null – generate the void object of type DOM_NULL

null() returns the void object of domain type DOM_NULL.

Call(s):

- null()

Return Value: the void object of domain type DOM_NULL.

Related Functions: _exprseq, _stmtseq, FAIL, NIL

Details:

- null() returns the only object of domain type DOM_NULL. It represents an empty sequence of MuPAD expressions or statements.
- The void object does not produce any output on the screen.
- Various systems functions such as print or reset return the void object.
- The void object is removed from sequences ("flattening"). It can be used to remove elements from lists or sets. Cf. example 2.
- null is a function of the system kernel.

Example 1. null() returns the void object which does not produce any screen output:

```
>> null()
```

The resulting object is of domain type DOM_NULL:

```
>> domtype(null())
```

DOM_NULL

This object represents the empty expression sequence and the empty statement sequence:

```
>> domtype(_exprseq()), domtype(_stmtseq())
```

```
DOM_NULL, DOM_NULL
```

Some system functions such as `print` return the void object:

```
>> print("Hello world!"):

                    "Hello world!"

>> domtype(%)

                    DOM_NULL
```

Example 2. The void object is removed from lists, sets, and expression sequences:

```
>> [null(), a, b, null(), c], {null(), a, b, null(), c},
   f(null(), a, b, null(), c)

          [a, b, c], {a, b, c}, f(a, b, c)

>> a + null() + b = _plus(a, null(), b)

                    a + b = a + b

>> subsop([a, x, b], 2 = null()), subs({a, x, b}, x = null())

                    [a, b], {a, b}
```

However, `null()` is a valid entry in arrays and tables:

```
>> a := array(1..2): a[1] := 1: a[2] := null(): a

                    +-          -+
                    | 1, null() |
                    +-          -+

>> domtype(a[1]), domtype(a[2])

                    DOM_INT, DOM_NULL

>> t := table(null() = "void", 1 = 2.5, b = null())

                    table(
                      b = null(),
                      1 = 2.5,
                      null() = "void"
                    )

>> domtype(t[b]), t[]

                    DOM_NULL, "void"

>> delete a, t:
```

Example 3. The void object remains if you delete all elements from an expression sequence:

```
>> a := (1, b): delete a[1]: delete a[1]: domtype(a)
```

```
DOM_NULL
```

The operand function `op` returns the void object when applied to an object with no operands:

```
>> domtype(op([])), domtype(op({})), domtype(op(f()))
```

```
DOM_NULL, DOM_NULL, DOM_NULL
```

```
>> delete a:
```

`numer` – the numerator of a rational expression

`numer(f)` returns the numerator of the expression `f`.

Call(s):

- `numer(f)`

Parameters:

`f` — an arithmetical expression

Return Value: an arithmetical expression.

Overloadable by: `f`

Related Functions: `denom`, `factor`, `gcd`, `normal`

Details:

- `numer` regards the input as a rational expression: non-rational subexpressions such as `sin(x)`, `x^(1/2)` etc. are internally replaced by "temporary variables". The numerator of this rationalized expression is computed, the temporary variables are finally replaced by the original subexpressions.
- Numerator and denominator are not necessarily cancelled: the numerator returned by `numer` may have a non-trivial `gcd` with the denominator returned by `denom`. Preprocess the expression by `normal` to enforce cancellation of common factors. Cf. example 2.

Example 1. We compute the numerators of some expressions:

```
>> numer(-3/4)

                                   -3

>> numer(x + 1/(2/3*x -2/x))

                                  3
                               2 x  - 3 x

>> numer((cos(x)^2 -1)/(cos(x) -1))

                                    2
                             cos(x)  - 1
```

Example 2. `numer` performs no cancellations if the rational expression is of the form "numerator/denominator":

```
>> r := (x^2 - 1)/(x^3 - x^2 + x - 1): numer(r)

                                  2
                                 x  - 1
```

This numerator has a common factor with the denominator of `r`; `normal` enforces cancellation of common factors:

```
>> numer(normal(r))

                                 x + 1
```

However, automatic normalization occurs if the input expression is a sum:

```
>> numer(r + x/(x + 1) + 1/(x + 1) - 1)

                                 x + 1

>> delete r:
```

`ode` – the domain of ordinary differential equations

`ode(eq, y(x))` represents an ordinary differential equation (ODE) for the function `y(x)`.

`ode({eq1, eq2, ...}, {y1(x), y2(x), ...})` represents a system of ODEs for the functions `y1(x)`, `y2(x)` etc.

Call(s):

- `ode(eq, y(x))`
- `ode({eq <, inits>}, y(x))`
- `ode({eq1, eq2, ... <, inits>}, {y1(x), y2(x), ...})`

Parameters:

`eq, eq1, eq2, ...`	—	equations or arithmetical expressions in the unknown functions and their derivatives with respect to `x`. An arithmetical expression is regarded as an equation with vanishing right hand side.
`y, y1, y2, ...`	—	the unknown functions: identifiers
`x`	—	the independent variable: an identifier
`inits`	—	the initial or boundary conditions: a sequence of equations

Return Value: an object of type `ode`.

Further Documentation: Section 8.3 of the Tutorial.

Related Functions: `numeric::odesolve`, `numeric::odesolve2`, `plot::ode`

Details:

- In the equations `eq`, `eq1` etc., the unknown functions must be represented by `y(x)`, `y1(x)` etc. Derivatives may be represented either by the `diff` function or by the differential operator `D`. Note that the token `'` provides a handy short cut: `y'(x) = D(y)(x)` $\equiv$ `diff(y(x), x)`.

- The unknown functions must be univariate in the independent variable `x`. Multivariate expressions such as `y(x, t)` are not accepted.

- Initial and boundary conditions are defined by sequences of equations involving the unknown functions or their derivatives on the left hand side. The corresponding values must be specified on the right hand side of the equations. In particular, the differential operator `D` (or the token `'`) must be used to specify values of derivatives at some point. E.g.,

  `y(1) = 2, y'(0) = 0, y■(0) = 1`

  is a valid sequence of boundary conditions for `inits`.

  Boundary conditions of the first and second kind are allowed. Mixed conditions are not accepted.

  The initial/boundary points and the corresponding initial/boundary values may be symbolic expressions.

- For scalar initial value or boundary value problems, use `ode({eq, inits}, y(x))` to specify the conditions.
- For systems of ODEs, there must be as many equations as unknown functions.
- The main purpose of the `ode` domain is to provide an environment for overloading the function `solve`.

  In the case of one single equation (possibly together with initial or boundary conditions), `solve` returns a set of explicit solutions or an implicit solution. Each element of the set represents a solution branch.

  In the case of a system of equations, `solve` returns a set of lists of equations for the unknown functions. Each list represents a solution branch.

  An symbolic `solve` call is returned if no solution is found.
- After `setuserinfo(ode, 10)`, a `solve` command provides information on MuPAD's way of solving ODEs.

Example 1. In the following, we show how to create and solve a scalar ODE. First, we define the ODE $x^2\, y'(x) + 3\, x\, y(x) = \sin(x)/x$. We use the quote token ' to represent derivatives:

```
>> eq := ode(x^2*y'(x) + 3*x*y(x) = sin(x)/x, y(x))

             /            sin(x)    2                      \
         ode| 3 x y(x) - ------ + x  diff(y(x), x), y(x) |
             \               x                             /
```

We get an element of the domain `ode` which we can now solve:

```
>> solve(eq)

                                {   C1 + cos(x) }
                                { - ----------- }
                                {         3     }
                                {        x      }

>> delete eq:
```

Example 2. An initial value problem is defined as a set consisting of the ODE and the initial conditions:

```
>> ivp := ode({f''(t) + 4*f(t) = sin(2*t),
               f(0) = a, f'(0) = b}, f(t))
```

```
 ode({f(0) = a, D(f)(0) = b, 4 f(t) + diff(f(t), t, t) -

    sin(2 t)}, f(t))
```

```
>> solve(ivp)
```

```
       { sin(2 t)                b sin(2 t)   t cos(2 t) }
       { -------- + a cos(2 t) + ---------- - ---------- }
       {     8                        2             4     }
```

```
>> delete ivp:
```

Example 3. With some restrictions, it is also possible to solve systems of ODEs. First, we define a system:

```
>> sys := {x'(t) - x(t) + y(t) = 0, y'(t) - x(t) - y(t) = 0}

     {y(t) - x(t) + D(x)(t) = 0, D(y)(t) - y(t) - x(t) = 0}
```

A call to `solve` yields the general solution with arbitrary parameters:

```
>> solution := solve(ode(sys, {x(t), y(t)}))

 {[y(t) = C7 exp((1 + I) t) + C8 exp((1 - I) t),

    x(t) = I C7 exp((1 + I) t) - I C8 exp((1 - I) t)]}
```

To verify the result, we substitute it back into the system `sys`. However, for the substitution, it is necessary to rewrite the system into a notation using the `diff` function:

```
>> eval(subs(rewrite(sys, diff), op(solution)))

                              {0 = 0}
```

```
>> delete sys, solution:
```

Example 4. In this example, we point out the various return formats of `ode`'s solve facility. First, we solve an ODE with an initial condition. The solution involves a symbolic integral:

```
>> solve(ode({y'(x) + x*y(x) = cos(x), y(0) = 3}, y(x)))

     {      /    2 \                        2 1/2            }
     {      |   x  |   int(cos(t4) exp(t4 )    , t4 = 0..x) }
     { 3 exp| - -- | + ---------------------------------- }
     {      \    2 /                        2 1/2            }
     {                              exp(x )                  }
```

The following system is solved incompletely:

```
>> sys := {x'(t) = -3*y(t)*z(t),
           y'(t) =  3*x(t)*z(t),
           z'(t) = -x(t)*y(t)}:
   solution := solve(ode(sys, {x(t), y(t), z(t)}))

                               2 1/2                            2 1/2
 {[x(t) = (C10 - C11 + 3 z(t) )    , y(t) = (- C10 - 3 z(t) )    ]

                                   2 1/2
     , [x(t) = (C10 - C11 + 3 z(t) )    ,

                              2 1/2
    y(t) = - (- C10 - 3 z(t) )    ],

                                   2 1/2
    [x(t) = - (C10 - C11 + 3 z(t) )    ,

                            2 1/2
    y(t) = (- C10 - 3 z(t) )    ], [

                                  2 1/2
    x(t) = - (C10 - C11 + 3 z(t) )    ,

                              2 1/2
    y(t) = - (- C10 - 3 z(t) )    ]}
```

In these four different solutions branches, no solution for the unknown function `z(t)` is provided. In fact, the partial solution above is subject to a further condition on `z(t)`. We substitute the first of our four solutions back into the system `sys`:

```
>> eval(subs(rewrite(sys, diff), solution[1]))

 {                                  2 1/2
 { diff(z(t), t) = - (- C10 - 3 z(t) )
 {
 {

                         2 1/2    3 z(t) diff(z(t), t)
    (C10 - C11 + 3 z(t) )    , - -------------------- =
                                                2 1/2
                                 (- C10 - 3 z(t) )

                                2 1/2
    3 z(t) (C10 - C11 + 3 z(t) )    ,
```

```
    3 z(t) diff(z(t), t)                                 2 1/2 }
  ----------------------- = - 3 z(t) (- C10 - 3 z(t) )    }
                     2 1/2                                     }
  (C10 - C11 + 3 z(t) )                                        }
```

For each solution branch, there remains is exactly one differential equation for z(t) to solve.

```
>> delete sys, solution:
```

Example 5. It may happen that MuPAD cannot solve a given equation. In such a case, a symbolic solve command is returned:

```
>> solve(ode(y'(x) + y(x)^2 = b + a*x, y(x)))

                                                       2
       solve(ode(- b - a x + diff(y(x), x) + y(x)  , y(x)))
```

Example 6. MuPAD's ODE solver contains algebraic algorithms for computing Liouvillian solutions of linear ordinary differential equations over the rational functions as well. These algorithms are based on differential Galois theory. For second order equations, an algorithm similar to the Kovacic algorithm is implemented. However, instead of computing the so-called Riccati-polynomial of a solution, this algorithm computes the solutions directly using formulas. Only when both solutions are algebraic functions, then in three cases it is necessary to compute the minimal polynomial of a solution, again using formulas. Hence, for Kovacic's famous example

$$y'' + \left(\frac{3}{16x^2} + \frac{2}{9(x-1)^2} - \frac{3}{16x(x-1)}\right) y = 0,$$

it is possible to compute the minimal polynomial of solution:

```
>> solve(ode(y''(x) + (3/(16*x^2) + 2/(9*(x - 1)^2)
                       - 3/(16*x*(x - 1)))*y(x), y(x)))

                24             8        8
 {C13 RootOf(_Y1   + 2985984 x  (x - 1)  -

              4    8        6          2    16        3
    2799360 x  _Y1  (x - 1)  - 4320 x  _Y1   (x - 1)  -

             5    4                  1/2      1/2 7
    165888 x  _Y1  (x - 2) (I x 3    - I 3   )  +

             3    12  1/2                 1/2      1/2 4
    5760 I x  _Y1   3    (x - 2) (I x 3    - I 3   ) , _Y1)}
```

MuPAD may find Liouvillian solutions for higher order equations as well. However, there is no guarantee that all of them are found. The following third order equation can be solved completely:

```
>> solve(ode(diff(y(x), x, x, x)
           + diff(y(x), x, x)*(2*x^2 - 1)/(2*x^2 - 2*x)
           + diff(y(x), x)*(295*x - 491*x^2 + 196*x^3 - 98)
                            /(196*x^2 - 392*x^3 + 196*x^4)
           + y(x)*(3*x - x^2 - 1)
                   /(196*x^2 - 392*x^3 + 196*x^4), y(x)))

 {
 {                         1/2     1/14
 { C17 (2 x + 2 (x (x - 1))    - 1)     +
 {
 {
 {
 {

                  C18
    ------------------------------- +
                         1/2     1/14
    (2 x + 2 (x (x - 1))    - 1)

    /
    |                           1/2     1/14
    | C16 (2 x + 2 (x (x - 1))    - 1)
    |
    \

    /
    |                         1/2
    | int(exp(-x) (x (x - 1))
    |
    \

                         1/2     1/14
    (2 x + 2 (x (x - 1))    - 1)    , x) /

                         1/2     1/7
    (2 x + 2 (x (x - 1))    - 1)    -

       /                          1/2         \ \ \      }
       |      exp(-x) (x (x - 1))             | | |      }
    int| ------------------------------, x | | | / 28 }
       |                         1/2     1/14    | | |      }
       \ (2 x + 2 (x (x - 1))    - 1)        / / /      }
                                                         }
```

```
}
```

Background:

- The implemented solution methods mainly stem from

  - Daniel Zwillinger. Handbook of differential equations. San Diego: Academic Press (1992).

  The algebraic algorithms for solving linear ODEs are described in

  - Winfried Fakler. Algebraische Algorithmen zur Lösung von linearen Differentialgleichungen. Stuttgart, Leipzig: Teubner, Reihe MuPAD Reports (1999).
  - Winfried Fakler. On second order homogeneous linear differential equations with Liouvillian solutions. Theor. Comp. Science **187**, 27-48 (1997).

`op` – the operands of an object

`op(object)` returns all operands of the object.

`op(object, i)` returns the `i`-th operand.

`op(object, i..j)` returns the `i`-th to `j`-th operands.

Call(s):

- `op(object)`
- `op(object, i)`
- `op(object, i..j)`
- `op(object, [i1, i2, ...])`

Parameters:

`object`	—	an arbitrary MuPAD object
`i, j`	—	nonnegative integers
`i1, i2, ...`	—	nonnegative integers or ranges of such integers

Return Value: a sequence of operands or the requested operand. `FAIL` is returned if no corresponding operand exists.

Overloadable by: `object`

Related Functions: `_index`, `contains`, `extnops`, `extop`, `extsubsop`, `map`, `new`, `nops`, `select`, `split`, `subsop`, `zip`

Details:

- MuPAD objects are composed of simpler parts: the "operands". The function `op` is the tool to decompose objects and to extract individual parts. The actual definition of an operand depends on the type of the object. The 'Background' section below explains the meaning for some of the basic data types.

- `op(object)` returns a sequence of all operands except the 0-th one. This call is equivalent to `op(object, 1..nops(object))`. Cf. example 1.

- `op(object, i)` returns the `i`-th operand. Cf. example 2.

- `op(object, i..j)` returns the `i`-th to `j`-th operands as an expression sequence; `i` and `j` must be nonnegative integers with `i` smaller or equal to `j`. This sequence is equivalent to `op(object, k) $ k = i..j`. Cf. example 3.

- `op(object, [i1, i2, ...])` is an abbreviation for the recursive call `op(...op(op(object, i1), i2), ...)` if `i1, i2, ...` are integers.

  A call such as `op(object, [i..j, i2])` with integers `i < j` corresponds to `map(op(object, i..j), op, i2)`. Cf. example 4.

- `op` returns `FAIL` if the specified operand does not exist. Cf. example 5.

- Expressions of domain type `DOM_EXPR`, arrays and floating point intervals have a 0-th operand.

  - For expressions, this is "the operator" connecting the other operands. In particular, for symbolic function calls, it is the name of the function.

  - For an array, the 0-th operand is a sequence consisting of an integer (the dimension of the array) and a range for each array index.

  - For a floating point interval, the value of the 0-th operand depends on the precise type of the interval: If the interval is a union of rectangles, the 0-th operand is `hold(_union)`. If the interval is not a union and consists only of real numbers, the 0-th operand is hold(hull). In the remaining case of a rectangle with non-vanishing imaginary part, the 0-th operand is `FAIL`.

  Other basic data types such as lists or sets do not have a 0-th operand. Cf. example 6.

- For library domains, `op` is overloadable. In the `"op"` method, the internal representation can be accessed with `extop`. It is sufficient to handle the cases `op(x)`, `op(x, i)`, and `op(x, i..j)` in the overloading method, the call `op(x, [i1, i2, ...])` needs not be considered. Cf. example 7.
- `op` is not overloadable for kernel domains.
- `op` is a function of the system kernel.

Example 1. The call `op(object)` returns all operands:

```
>> op([a, b, c, [d, e], x + y])

                     a, b, c, [d, e], x + y

>> op(a + b + c^d)

                                d
                         a, b, c

>> op(f(x1, x2, x3))

                          x1, x2, x3
```

Example 2. The call `op(object, i)` extracts a single operand:

```
>> op([a, b, c, [d, e], x + y], 4)

                            [d, e]

>> op(a + b + c^d, 3)

                              d
                             c

>> op(f(x1, x2, x3), 2)

                             x2
```

Example 3. The call `op(object, i..j)` extracts a range of operands:

```
>> op([a, b, c, [d, e], x + y], 3..5)

                        c, [d, e], x + y

>> op(a + b + c^d, 2..3)
```

d

b, c

```
>> op(f(x1, x2, x3), 2..3)
```

x2, x3

A range may include the 0-th operand if it exists:

```
>> op(a + b + c^d, 0..2)
```

_plus, a, b

```
>> op(f(x1, x2, x3), 0..2)
```

f, x1, x2

Example 4. The call `op(object, [i1, i2, ...])` specifies suboperands:

```
>> op([a, b, c, [d, e], x + y], [4, 1])
```

d

```
>> op(a + b + c^d, [3, 2])
```

d

```
>> op(f(x1, x2, x3 + 17), [3, 2])
```

17

Also ranges of suboperands can be specified:

```
>> op([a, b, c, [d, e], x + y], [4..5, 2])
```

e, y

```
>> op(a + b + c^d, [2..3, 1])
```

b, c

```
>> op(f(x1, x2, x3 + 17), [2..3, 1])
```

x2, x3

Example 5. Nonexisting operands are returned as `FAIL`:

```
>> op([a, b, c, [d, e], x + y], 8), op(a + b + c^d, 4),
   op(f(x1, x2, x3), 4)
```

FAIL, FAIL, FAIL

Example 6. For expressions of type `DOM_EXPR`, the 0-th operand is "the operator" connecting the other operands:

```
>> op(a + b + c, 0), op(a*b*c, 0), op(a^b, 0), op(a[1, 2], 0)

                  _plus, _mult, _power, _index
```

For symbolic function calls, it is the name of the function:

```
>> op(f(x1, x2, x3), 0), op(sin(x + y), 0), op(besselJ(0, x), 0)

                         f, sin, besselJ
```

The 0-th operand of an array is a sequence consisting of the dimension of the array and a range for each array index:

```
>> op(array(3..100), 0)

                            1, 3..100
```

```
>> op(array(1..2, 1..3, 2..4), 0)

                       3, 1..2, 1..3, 2..4
```

No 0-th operand exists for other kernel domains:

```
>> op([1, 2, 3], 0), op({1, 2, 3}, 0), op(table(1 = y), 0)

                        FAIL, FAIL, FAIL
```

Example 7. For library domains, `op` is overloadable. First, a new domain `d` is defined via `newDomain`. The `"new"` method serves for creating elements of this type. The internal representation of the domain is a list of all arguments of this `"new"` method:

```
>> d := newDomain("d"):  d::new := () -> new(dom, [args()]):
```

The `"op"` method of this domain is defined. It is to return the elements of a sorted copy of the internal list which is accessed via `extop`:

```
>> d::op := proc(x, i = null())
              local internalList;
            begin
              internalList := extop(x, 1);
              op(sort(internalList), i)
            end_proc:
```

By overloading, this method is called when the operands of an object of type `d` are requested via `op`:

```
>> e := d(3, 7, 1):  op(e);  op(e, 2);  op(e, 1..2)
```

$$1, 3, 7$$

$$3$$

$$1, 3$$

```
>> delete d, e:
```

Example 8. Identifiers, integers, real floating point numbers, character strings, and the Boolean constants are "atomic" objects. The only operand is the object itself:

```
>> op(x), op(17), op(0.1234), op("Hello World!")
```

    x, 17, 0.1234, "Hello World!"

For rational numbers, the operands are the numerator and the denominator:

```
>> op(17/3)
```

    17, 3

For complex numbers, the operands are the real part and the imaginary part:

```
>> op(17 - 7/3*I)
```

    17, -7/3

Example 9. For sets, `op` returns the elements according to the *internal* order. Note that this order may differ from the ordering with which sets are printed on the screen:

```
>> s := {1, 2, 3}
```

    {1, 2, 3}

```
>> op(s)
```

    3, 2, 1

Indexed access to set elements uses the ordering visible on the screen:

```
>> s[1], s[2], s[3]
```

    1, 2, 3

Note that access to set elements via `op` is much faster than indexed calls:

```
>> s := {sqrt(i) $ i = 1..500}:
   time([op(s)])/time([s[i] $ i = 1..nops(s)]);

                                  1/364

>> delete s:
```

Example 10. The operands of a list are its entries:

```
>> op([a, b, c, [d, e]])

                             a, b, c, [d, e]

>> op([[a11, a12], [a21, a22]], [2, 1])

                                   a21
```

Example 11. Internally, the operands of an array form a "linear" sequence containing all entries:

```
>> op(array(1..2, 1..2, [[11, 12], [21, 22]]))

                             11, 12, 21, 22
```

Undefined entries are returned as `NIL`:

```
>> op(array(1..2, 1..2))

                           NIL, NIL, NIL, NIL
```

Example 12. The operands of a table consist of equations relating the indices and the corresponding entries:

```
>> T := table((1, 2) = x + y, "diff(sin)" = cos, a = b)

                          table(
                            a = b,
                            "diff(sin)" = cos,
                            (1, 2) = x + y
                          )

>> op(T)

            a = b, "diff(sin)" = cos, (1, 2) = x + y

>> delete T:
```

Example 13. Expression sequences are not flattened:

```
>> op((a, b, c), 2)
```

```
                                    b
```

Note, however, that the arguments passed to `op` are evaluated. In the following call, evaluation of `x` flattens this object:

```
>> x := hold((1, 2), (3, 4)):  op(x, 1)
```

```
                                    1
```

Use `val` to prevent simplification of `x`:

```
>> op(val(x), 1)
```

```
                                  1, 2
```

```
>> delete x:
```

Background:

We explain the meaning of "operands" for some basic data types:

- Identifiers, integers, real floating point numbers, character strings, as well as the Boolean constants are "atomic" objects. They have only one operand: the object itself. Cf. example 8.
- A rational number of type `DOM_RAT` has two operands: the numerator and the denominator. Cf. example 8.
- A complex number of type `DOM_COMPLEX` has two operands: the real part and the imaginary part. Cf. example 8.
- The operands of a set are its elements.

  NOTE

  Note that the ordering of the elements as printed on the screen does not necessarily coincide with the internal ordering referred to by `op`. Cf. example 9.
- The operands of a list are its elements. Cf. example 10.
- The operands of arrays are its entries. Undefined entries are returned as `NIL`. Cf. the examples 11 and 6.
- The operands of tables are the equations associating an index with the corresponding entry. Cf. example 12.
- The operands of an expression sequence are its elements. Note that such sequences are not flattened by `op`. Cf. example 13.

- The operands of a symbolic function call such as `f(x, y, ...)` are the arguments `x`, `y` etc. The function name `f` is the 0-the operand.
- In general, the operands of expressions of type `DOM_EXPR` are given by their internal representation. There is a 0-th operand ("the operator") corresponding to the type of the expression. Internally, the operator is a system function, the expression corresponds to a function call. E.g., `a + b + c` has to be interpreted as `_plus(a, b, c)`, a symbolic indexed call such as `A[i, j]` corresponds to `_index(A, i, j)`. The name of the system function is the 0-th operand (i.e., `_plus` and `_index` in the previous examples), the arguments of the function call are the further operands.

`operator` – define a new operator symbol

`operator(symb, f, T, prio)` defines a new operator symbol `symb` of type `T` with priority `prio`. The function `f` evaluates expressions using the new operator.

`operator(symb, `*`Delete`*`)` removes the definition of the operator symbol `symb`.

Call(s):

- `operator(symb, f <, T, prio>)`
- `operator(symb, `*`Delete`*`)`

Parameters:

`symb`	—	the operator symbol: a character string.
`f`	—	the function evaluating expressions using the operator.
`T`	—	the type of the operator: one of the options *Prefix*, *Postfix*, *Binary* or *Nary*. The default is *Nary*.
`prio`	—	the priority of the operator: an integer between 1 and 1999. The default is 1300.

Options:

Prefix	—	the operator is a unary operator with prefix notation
Postfix	—	the operator is a unary operator with postfix notation
Binary	—	the operator is a non-associative binary operator with infix notation
Nary	—	the operator is an associative binary operator with infix notation
Delete	—	the operator with symbol `symb` is deleted

Return Value: the void object of type `DOM_NULL`.

Side Effects: The new operator symbol `symb` is known by the parser and may be used to enter expressions. The new operator symbol will *not* be used when reading files using the function `read` with the option `Plain`.

Details:

- `operator` is used to define new user-defined operator symbols or to delete them.
- Given the operator symbol `"++"`, say, with evaluating function `f`, the following expressions are built by the parser, depending on the type of the operator:

  `Prefix`: The input `++x` results in `f(x)`.

  `Postfix`: The input `x++` results in `f(x)`.

  `Binary`: The input `x ++ y ++ z` results in `f(f(x, y), z)`.

  `Nary`: The input `x ++ y ++ z` results in `f(x, y, z))`.

- There may exist operator symbols which are prefixes of other operator symbols. The scanner reads as many characters as possible and chooses the longest matching operator symbol. Cf. example 3.
- It is not possible to define two operators with the same symbol. So one may not define a unary `++` and a binary `++` at the same time.
- The following restrictions exist for the operator symbol string `symb`:

  - It may not be longer than 32 characters.
  - It may not start with a white-space.
  - It may not start with a \ (backslash) character.

  Thus, the strings `" @"` and `"\\/"` are not allowed. Please note that currently `operator` does not check these restrictions.

- Builtin operators may be redefined.
- It is not possible to define out-fix operators like $|x|$ or 3-nary or other types of operators.
- The new operator symbol is also used if files are read, with one exception: if a file is read with the function `read` using the option `Plain`, the new operator is not taken into account. (This option is used if MuPAD library files are read, because otherwise user-defined operators could change the meaning of the source code in an uncontrolled way.)

- Currently, there is no comfortable way to configure the output of expressions containing user-defined operators. (One may use the function `builtin` to define the text output of expressions. This, however, is not recommended.)
- See the MuPAD 2.0 quick reference for the precedence of the builtin operators.

Option <`Prefix`>:

- The operator is regarded as a unary operator with prefix notation. Given the operator symbol `"++"` and the evaluation function `f`, the input `++x` is parsed as the expression `f(x)`.

Option <`Postfix`>:

- The operator is regarded as a unary operator with postfix notation. Given the operator symbol `"++"` and the evaluation function `f`, the input `x++` is parsed as the expression `f(x)`.

Option <`Binary`>:

- The operator is regarded as a non-associative binary operator with infix notation. Given the operator symbol `"++"` and the evaluation function `f`, the input `x ++ y ++ z` is parsed as the expression `f(f(x, y), z)`, i.e. the operator binds left-to-right.

Option <`Nary`>:

- The operator is regarded as an associative n-ary operator with infix notation. Given the operator symbol `"++"` and the evaluation function `f`, the input `x ++ y ++ z` is parsed as the expression `f(x, y, z)`.

Example 1. This example shows how to define an operator symbol for the bit-shift operation (as in the language `C`):

```
>> bitshiftleft := (a, b) -> a * 2^b:
   operator("<<", bitshiftleft, Binary, 950):
```

After this call, the symbol ■ can be used to enter expressions:

```
>> 2 << 1, x << y
```

```
                                    y
                               4, x 2
```

```
>> operator("<<", Delete):
```

Example 2. Identifiers may be used as operator symbols:

```
>> operator("x", _vector_product, Binary, 1000):
```

```
>> a x b x c
```

```
            _vector_product(_vector_product(a, b), c)
```

```
>> operator("x", Delete):
```

Example 3. This example shows that the scanner tries to match the longest operator symbol:

```
>> operator("~", F, Prefix, 1000):
   operator("~>", F1, Prefix, 1000):
   operator("~~>", F2, Prefix, 1000):
```

```
>> ~~ x, ~~> x, ~ ~> x, ~~~> x
```

```
               F(F(x)), F2(x), F(F1(x)), F(F2(x))
```

```
>> operator("~", Delete):
   operator("~>", Delete):
   operator("~~>", Delete):
```

Background:

- When the scanner reads a new token, it first discards any whitespace and backslash characters. Then it tries to match user-defined operator symbols. The longest user-defined operator symbol matching the scanned characters is made the next token. If no user-defined operator symbol matches, it scans for the built-in tokens.
- The parser uses both recursive-descend and a operator precedence parsing. Built-in and user-defined operators are parsed using operator precedence.

`package` – load a package of new library functions

`package(dirname)` loads a new library package.

Call(s):

- `package(dirname <, Quiet> <, Forced>)`

Parameters:

`dirname` — a valid directory path: a character string

Options:

Quiet — suppresses screen output while loading the library
Forced — enforces reloading of libraries that are already loaded

Return Value: the value of the last statement in the initialization file `init.mu` of the package.

Side Effects: The path `dirname/lib` is *prepended* to the search path `LIBPATH`. The path `dirname/modules/OSName` is *prepended* to the search path `READPATH` (`OSName` is the name of the operating system; cf. `sysname`). This way, library functions are first searched for in the package. Modules contained in the package are found automatically. In case of a naming conflict, a package function overrides a function of the system's main library.

Related Functions: `export`, `FILEPATH`, `LIBPATH`, `loadmod`, `loadproc`, `newDomain`, `PACKAGEPATH`, `read`, `READPATH`

Details:

- In MuPAD, procedures implementing algorithms from a specific mathematical area are organized as libraries. E.g., `numlib` is the library for number theory, `numeric` is the library for numerical algorithms etc. Also the user should organize collections of related functions as a library package. With a suitable structure of the folder containing the files with the source code, the whole library can be loaded into the MuPAD session via a call to `package`.

- Formally, a library is a domain. The functions in the library are its slots and are accessed by the "slot operator" `::` as in `numlib::fibonacci`, `numeric::int` etc.

- Typically, either a new library domain is to be created and its functions are to be loaded by `package`, or new functions are to be added to an existing library domain of MuPAD's standard installation. The detailed example 1 below is devoted to the former case, whereas example 2 covers the latter case. Special care should be taken, when existing libraries are modified: the user should make sure that existing functionality is not overwritten or destroyed by the modification.

- The folder `mypack`, say, containing the library package to be loaded can be placed anywhere in the filesystem. The pathname specified in a `package` call may be an absolut path (from the root to `mypack`). Alternatively, a path relative to the "working directory" may be specified.

  Note that the "working directory" is different on different operating systems. On Windows systems, for example, the "working directory" is the folder, where MuPAD is installed. On UNIX or Linux systems, it is the directory in which the current MuPAD session was started.

  If the environment variable `PACKAGEPATH` contains the path to the folder `mypack`, `package` only needs the name of the package as its argument, which is `"mypack"`.

- The folder `mypack` must have the same hierarchical structure as the standard MuPAD library. In particular, it must have a subfolder `lib` containing the source files of the package. Inside the `lib` folder, an initialization file `init.mu` must exist.

  For example, on a UNIX or Linux system, the folder `mypack` should have the following structure (up to different path separators, the same holds for other operating systems as well):

  ```
  mypack/lib/init.mu
  mypack/lib/LIBFILES/mylib.mu
  mypack/lib/MYLIB/stuff.mu
  mypack/lib/MYLIB/...
  mypack/lib/MYLIB/SUBDIR/morestuff.mu
  mypack/lib/MYLIB/SUBDIR/...
  ```

  Typically, the initialization file `init.mu` uses `loadproc` commands to define the objects (new library domains and/or functions) of the package.

  If a new library domain is to be created, the `lib` folder should contain a subfolder `LIBFILES` with a file `LIBFILES/mylib.mu`. The `loadproc` commands inside `init.mu` should refer to the file `mylib.mu`. Inside this file, the new library domain should be created via `newDomain`. The functions (slots) of this new library domain should again be declared via `loadproc` commands that refer to the actual location of the files containing the source code of these functions. The code files should be organized in folders such as `lib/MYLIB`, `lib/MYLIB/SUBDIR` etc.

  This structure and the loading mechanism corresponds to the organization of MuPAD's main library. It uses the initialization file `MuPAD_ROOT_PATH/lib/sysinit.mu`.

- If a new library domain `mylib`, say, is to be generated by the package, the initialization file `mypack/lib/init.mu` should refer to the file `LIBFILES/mylib.mu` where the library is actually created:

  ```
  // -------- file mypack/lib/init.mu --------
  ```

```
// load the library domain 'mylib'
alias(path = pathname("LIBFILES")):
mylib := loadproc(mylib, path, "mylib"):
unalias(path):
stdlib::LIBRARIES := stdlib::LIBRARIES union {"mylib"}:
// The return value of the package call:
null():
// ---------- end of file init.mu ----------
```

By adding the new library domain `mylib` to the set `stdlib::LIBRARIES`, a call to `package` will automatically launch the `info` function to print information about the new package. The information includes the string `mylib::info` that should be defined in `LIBFILES/mylib.mu`.

The value of the last statement in the file `init.mu` is the return value of a `package` call. Typically, this is the `null()` object to avoid any unwanted screen output when loading the package. Alternatively, some useful information such as the string `"package 'mylib' successfully loaded"` may be returned.

Cf. example 1 for further details.

- The file `LIBFILES/mylib.mu` should generate the new library domain via `newDomain`. Some standard entries such as `mylib::Name`, `mylib::info`, and `mylib::interface` should be defined. The functions `mylib::function1` etc. of the new library should refer to the actual code files via `loadproc`:

```
// ---- file mypack/lib/LIBFILES/mylib.mu ----
// mylib -- a library containing my functions
mylib := newDomain("mylib"):
mylib::Name := "mylib":
mylib::info := "Library 'mylib': a library with my functions":
mylib::interface := {hold(function1), hold(function2), ...}:
// define the functions implemented in ../MYLIB/function1.mu etc:
alias(path = pathname("MYLIB")):
mylib::function1 := loadproc(mylib::function1, path, "function1"):
mylib::function2 := loadproc(mylib::function2, path, "function2"):
...
unalias(path):
// define the functions implemented in ../MYLIB/SUBDIR/more1.mu etc:
alias(path = pathname("MYLIB", "SUBDIR")):
mylib::more1 := loadproc(mylib::more1, path, "more1"):
mylib::more2 := loadproc(mylib::more2, path, "more2"):
...
unalias(path):
null():
// -------- end of file mylib.mu ------------
```

Cf. example 1 for further details.

- When a package initialisation file is read the variable `FILEPATH` contains the path of the file.

Option <`Quiet`>:

- This option suppresses printing of information about the package during loading.

Option <`Forced`>:

- Usually, a package is loaded only once; a further attempt to reload the package causes an error. This option allows to enforce reloading of packages that are already loaded.

Example 1. In the following, we demonstrate how a package should be organized that generates a new library domain containing user-defined functions. In example 2, we load the same functions, but include them in one of MuPAD's standard libraries rather than create a new library domain.

Suppose we have implemented some functions operating on integers such as a factorial function and a new function for computing powers of integers. It is a good idea to combine these functions into one package. The new library domain is to be called `numfuncs` (for elementary number theoretic functions). It is organized as a package stored in the folder `demoPack1`. This folder has the following structure:

```
demoPack1/lib/init.mu
demoPack1/lib/LIBFILES/numfuncs.mu
demoPack1/lib/NUMFUNCS/factorial.mu
demoPack1/lib/NUMFUNCS/russian.mu
```

The initialization file `init.mu` may be implemented as follows:

```
// ----- file demoPack1/lib/init.mu -----
// loads the library 'numfuncs'
alias(path = pathname("LIBFILES")):
numfuncs := loadproc(numfuncs, path, "numfuncs"):
stdlib::LIBRARIES := stdlib::LIBRARIES union {"numfuncs"}:
unalias(path):
// return value of package:
"library 'numfuncs' successfully loaded":
// -------- end of file init.mu ---------
```

The function pathname is used to create the pathname in a form that is appropriate for the currently used operating system. The loadproc call refers to the actual definition of the new library domain in the file LIBFILES/numfuncs.mu:

```
// --- file demoPack1/lib/LIBFILES/numfuncs.mu ---
// numfuncs -- the library for elementary number theory
numfuncs := newDomain("numfuncs"):
numfuncs::Name := "numfuncs":
numfuncs::info := "Library 'numfuncs': the library of ".
                  "functions for elementary number theory":
numfuncs::interface := {hold(factorial), hold(russianPower)}:
// define the functions implemented in ../NUMFUNCS/factorial.mu etc:
alias(path = pathname("NUMFUNCS")):
numfuncs::factorial :=
       loadproc(numfuncs::factorial, path, "factorial"):
numfuncs::odd :=
       loadproc(numfuncs::odd, path, "russian"):
numfuncs::russianPower :=
       loadproc(numfuncs::russianPower, path, "russian"):
unalias(path):
null():
// --------- end of file numfuncs.mu ---------
```

Here, the new library domain is created via newDomain. Any library domain should have the entries Name and info. One may also define an interface entry, which is to contain all the functions a user should be aware of.

This file also contains the definitions of the functions factorial, odd, and russianPower which are implemented in the subfolder demoPack1/lib/NUMFUNCS. (See example 2 for details of the implementation; just replace numlib by numfuncs.)

The function numfuncs::factorial is implemented in a separate file. The functions numfuncs::odd and numfuncs::russianPower are both installed in the file russian.mu.

Note that numfuncs::odd is not added to the interface slot, because it is a utility function that should not be seen and used by the user.

Finally, we demonstrate the loading of the library package. Suppose that we have several packages, installed in the folder myMuPADFolder:

```
/home/myLoginName/myMuPADFolder/demoPack1
/home/myLoginName/myMuPADFolder/demoPack2
...
```

The library numfuncs installed in demoPack1 is loaded by a call to the package function:

```
>> PACKAGEPATH := "/home/myLoginName/myMuPADFolder/", PACKAGEPATH:
   package("demoPack1")
```

```
 Library 'numfuncs': the library of functions for elementary \
 number theory

 -- Interface:
 numfuncs::factorial, numfuncs::russianPower

           "library 'numfuncs' successfully loaded"
```

In the initialization file `init.mu`, the new library was added to `stdlib::LIBRARIES`. For the reason, loading causes the above information about the library to be printed. By default, a library package can be loaded only once:

```
>> package("demoPack1")

 Warning: Package already defined. For redefinition use option \
 Forced [package]
```

Following the warning, we overwrite the existing library `numfuncs` by another call to `package` using the option *`Forced`*:

```
>> package("demoPack1", Forced)

 Warning: Package redefined [package]

            "library 'numfuncs' successfully loaded"
```

After loading, the new library `numfuncs` is fully integrated into the system. Its functions can be called like any other function of MuPAD's main library:

```
>> numfuncs::factorial(41)

       33452526613163807108170062053440751665152000000000

>> numfuncs::russianPower(123, 12)

                   11991163848716906297072721
```

Example 2. We demonstrate how a package should be organized that adds new functions to an existing library domain.

We consider the same functions as in example 1. However, instead of creating a new library domain, we wish to add these functions to the existing library domain `numlib` of MuPAD's main library. In particular, the package is to install the new functions `numlib::factorial` and `numlib::russianPower`. Before loading such functions, we should make sure that they do not overwrite existing functions of the standard `numlib` installation. As a simple test to check that the standard installation does not provide a function `numlib::factorial`, one may simply try to call this function:

```
>> numlib::factorial
```

```
                         FAIL
```

Indeed, this function does not exist yet and shall now be provided by an extension installed in a folder `demoPack2`:

```
demoPack2/lib/init.mu
demoPack2/lib/NUMLIB/factorial.mu
demoPack2/lib/NUMLIB/russian.mu
```

In this case, no new library domain is to be created. Hence, in contrast to example 1, no file `demoPack2/lib/LIBFILES/numlib.mu` needs to be installed (which would be in conflict with the corresponding file defining the `numlib` library domain of the standard installation). Instead, the new functions may be declared directly in the initialization file `init.mu` as follows:

```
// ------ file demoPack2/lib/init.mu -------
// loads additional functions for the existing library 'numlib'
numlib::interface := numlib::interface
      union {hold(factorial), hold(russianPower)}:
// define the functions implemented in ../NUMLIB/factorial.mu etc:
alias(path = pathname ("NUMLIB")):
numlib::factorial :=
      loadproc(numlib::factorial, path, "factorial"):
numlib::odd :=
      loadproc(numlib::odd, path, "russian"):
numlib::russianPower :=
      loadproc(numlib::russianPower, path, "russian"):
unalias(path):
// return value of package:
"new numlib functions successfully loaded":
// ---------- end of file init.mu ----------
```

Similar to example 1, we added the main functions to the existing `interface` slot of `numlib`.

We now have a look into the files `factorial.mu` and `russian.mu` containing the source code of the functions:

```
// ---- file demoPack2/lib/NUMLIB/factorial.mu ----
numlib::factorial :=
  proc(n : Type::NonNegInt) : Type::PosInt
    // factorial(n) computes n!
  begin
    if n = 0 then 1
    else n*numlib::factorial(n - 1)
    end_if
  end_proc:
// -------- end of file factorial.mu ---------
```

The routine `numlib::odd` is a utility function for `numlib::russianPower`. Both functions are coded in one file:

```
// ---- file demoPack2/lib/NUMLIB/russian.mu ----
numlib::odd := m -> not(iszero(m mod 2)):

numlib::russianPower :=
  proc(m : DOM_INT, n : Type::NonNegInt) : DOM_INT
    // computes the n-th power of m using the
    // russian peasant method of multiplication
    local d;
  begin
    d := 1;
    while n>0 do
      if numlib::odd(n) then
        d := d*m;
        n := n - 1;
      else
        m := m*m;
        n := n div 2;
      end_if
    end_while;
    d
  end_proc:
// ----------- end of file russian.mu ------------
```

Finally, we demonstrate the loading of the functions. Suppose that we have several packages, installed in the folder `myMuPADFolder`:

```
/home/myLoginName/myMuPADFolder/demoPack1
/home/myLoginName/myMuPADFolder/demoPack2
...
```

The functions installed in `demoPack2` are loaded by a call to the `package` function:

```
>> PACKAGEPATH := "/home/myLoginName/myMuPADFolder", PACKAGEPATH:
   package("demoPack2")

           "new numlib functions successfully loaded"
```

The new functions added to the `interface` slot of `numlib` are listed by an `info` call:

```
>> info(numlib)
```

```
Library 'numlib': the package for elementary number theory

-- Interface:
numlib::Lambda,          numlib::Omega,
...
numlib::factorial,       numlib::fibonacci,
...
numlib::proveprime,      numlib::russianPower,
...
```

After loading, the new functions are fully integrated into the library and can be called like any other function of MuPAD's library:

```
>> numlib::factorial(41)

       33452526613163807108170062053440751665152000000000

>> numlib::russianPower(123, 12)

                   11991163848716906297072721
```

Changes:

- package now looks in PACKAGEPATH for packages.

---

pade – Pade approximation

pade(f, ..) computes a Pade approximant of the expression f.

Call(s):

- pade(f, x <, [m, n]>)
- pade(f, x = x0 <, [m, n]>)

Parameters:

- f — an arithmetical expression or a series of domain type Series::Puiseux generated by the function series
- x — an identifier
- x0 — an arithmetical expression. If x0 is not specified, then x0 = 0 is assumed.

Options:

- [m, n] — a list of nonnegative integers specifying the order of the approximation. The default values are [3, 3].

Return Value: an arithmetical expression or `FAIL`.

Related Functions: `series`

Details:

- The Pade approximant of order $[m, n]$ around $x = x_0$ is a rational expression
$$(x - x_0)^p \, \frac{a_0 + a_1 (x - x_0) + \cdots + a_m (x - x_0)^m}{1 + b_1 (x - x_0) + \cdots + b_n (x - x_0)^n}$$
approximating f. The parameters p and a_0 are given by the leading order term $f = a_0 (x - x_0)^p + O((x - x_0)^{p+1})$ of the series expansion of f around $x = x_0$. The parameters $a_1, \ldots, b_n$ are chosen such that the series expansion of the Pade approximant coincides with the series expansion of f to the maximal possible order.

- The expansion points `infinity`, `-infinity`, and `complexInfinity` are not allowed.

- If no series expansion of f can be computed, then `FAIL` is returned. Note that `series` must be able to produce a Taylor series or a Laurent series of f, i.e., an expansion in terms of integer powers of $x - x_0$ must exist.

Example 1. The Pade approximant is a rational approximation of a series expansion:

```
>> f := cos(x)/(1 + x): P := pade(f, x, [2, 2])

                     2
          2 x - 7 x  + 12
          ---------------
                     2
           14 x + x  + 12
```

For most expressions of leading order 0, the series expansion of the Pade approximant coincides with the series expansion of the expression through order $m + n$:

```
>> S := series(f, x, 6)

                   2    3       4        5
                  x    x    13 x     13 x        6
      1 - x + -- - -- + ----- - ----- + O(x )
                  2    2     24       24
```

This differs from the expansion of the Pade approximant at order 5:

```
>> series(P, x, 6)
```

```
                  2    3       4       5
                 x    x    13 x    85 x       6
         1 - x + -- - -- + ----- - ----- + O(x )
                  2    2     24      144
```

The series expansion can be used directly as input to pade:

```
>> pade(S, x, [2, 3]), pade(S, x, [3, 2])

                    2                  2      3
          12 - 5 x            12 x + 7 x  - 7 x  - 12
      -------------------, -----------------------
                  2    3                 2
      12 x + x  + x  + 12           13 x  - 12
```

Both Pade approximants approximate f through order $m + n = 5$:

```
>> map([%], series, x)

 --           2    3       4       5
 |           x    x    13 x    13 x       6
 |  1 - x + -- - -- + ----- - ----- + O(x ),
 --           2    2     24      24

              2    3       4       5          --
             x    x    13 x    13 x       6    |
    1 - x + -- - -- + ----- - ----- + O(x )   |
              2    2     24      24           --

>> delete f, P, S:
```

Example 2. The following expression does not have a Laurent expansion around $x = 0$:

```
>> series(x^(1/3)/(1 - x), x)

       1/3    4/3    7/3    10/3     13/3     16/3      19/3
      x    + x    + x    + x     + x     + x     + O(x    )
```

Consequently, pade fails:

```
>> pade(x^(1/3)/(1 - x), x, [3, 2])

                              FAIL
```

Example 3. Note that the specified orders $[m, n]$ do not necessarily coincide with the orders of the numerator and the denominator if the series expansion does not start with a constant term:

```
>> pade(x^10*exp(x), x, [2, 2]), pade(x^(-10)*exp(x), x, [2, 2])

                10      11    12               2
            12 x   + 6 x   + x        6 x + x  + 12
            --------------------, --------------------
                 2                      10      11    12
                x  - 6 x + 12      12 x   - 6 x   + x
```

---

partfrac – **compute a partial fraction decomposition**

partfrac(f, x) returns the partial fraction decomposition of the rational expression f with respect to the variable x.

Call(s):

- partfrac(f <, x>)

Parameters:

f — a rational expression in x
x — the indeterminate: typically, an identifier or an indexed identifier.

Return Value: an arithmetical expression.

Overloadable by: f

Further Documentation: Chapter "Manipulating Expressions" of the Tutorial.

Related Functions: collect, denom, divide, expand, factor, normal, numer, rectform, rewrite, simplify

---

Details:

- Consider the rational expression $f(x) = g(x) + p(x)/q(x)$ with polynomials g, p, q satisfying degree(p) < degree(q). Here, q = denom(f) is the denominator of f, and g, p, given by (g, p) = divide(numer(f), q, [x]), are the quotient and the remainder of the polynomial division of the numerator of f by the denominator q. Let

$$q(x) = q_1(x)^{e_1} \cdot q_2(x)^{e_2} \cdot \ldots$$

be a factorization of the denominator into nonconstant and pairwise coprime polynomials q_i with integer exponents e_i. The partial fraction decomposition based on this factorization is a representation

$$f(x) = g(x) + \frac{p_{11}(x)}{q_1(x)} + \cdots + \frac{p_{1e_1}(x)}{q_1(x)^{e_1}} + \frac{p_{21}(x)}{q_2(x)} + \cdots + \frac{p_{2e_2}(x)}{q_2(x)^{e_2}} + \cdots$$

with polynomials p_{ij} satisfying $\mathtt{degree}(p_{ij}) < \mathtt{degree}(q_i)$. In particular, the polynomials p_{ij} are constant if q_i is a linear polynomial.

`partfrac` uses the factors q_i of `q` = `denom(f)` found by the function `factor`. The factorization is computed over the field implied by the coefficients of the denominator (see `factor` for details). Cf. example 2.

- The second argument `x` in a call to `partfrac` can be omitted if `f` has only one indeterminate.

- The partial fraction decomposition can also be computed for expressions that are rational with respect to a symbolic function call. This function call must be specified as the indeterminate. Cf. example 3.

Example 1. In the following calls, there is no need to specify an indeterminate because the rational expressions are univariate:

```
>> partfrac(x^2/(x^3 - 3*x + 2))

                  5           1            4
              --------- + ---------- + ---------
              9 (x - 1)            2   9 (x + 2)
                          3 (x - 1)

>> partfrac(23 + (x^4 + x^3)/(x^3 - 3*x + 2))

                  19          2            8
          x + --------- + ---------- + --------- + 24
              9 (x - 1)            2   9 (x + 2)
                          3 (x - 1)
```

The following expression contains two indeterminates `x` and `y`. One has to specify the variable with respect to which the partial fraction decomposition shall be computed:

```
>> f := x^2/(x^2 - y^2): partfrac(f, x),  partfrac(f, y)

           y           y                 x           x
       --------- - --------- + 1, --------- + ---------
       2 (x - y)   2 (x + y)      2 (x + y)   2 (x - y)

>> delete f:
```

Example 2. In the following, we demonstrate the dependence of the partial fraction decomposition on the function `factor`:

```
>> partfrac(1/(x^2 + 2), x)

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

Note that the denominator $x^2 + 2$ does not factor over the rational numbers:

```
>> factor(x^2 + 2)

                                  2
                                 x  + 2
```

However, it factors over the extension containing $\sqrt{-2}$. In the following calls, this extended coefficient field is implicitly assumed by `factor` and, consequently, by `partfrac`:

```
>> factor(sqrt(-2)*x^2 + 2*sqrt(-2))

                  1/2          1/2           1/2
              (I 2   ) (x - I 2   ) (x + I 2   )

>> partfrac(x/(sqrt(-2)*x^2 + 2*sqrt(-2)), x)

                 1/2 /        1                1       \
        - 1/2 I 2    | -------------- + -------------- |
                     |            1/2              1/2  |
                     \ 2 (x - I 2   )   2 (x + I 2   ) /
```

Example 3. Rational expressions of symbolic function calls may also be decomposed into partial fractions:

```
>> partfrac(1/(sin(x)^4 - sin(x)^2 + sin(x) - 1), sin(x))

                                                  2
                                 2 sin(x)   sin(x)
                               - -------- - ------- - 2/3
                 1                  3          3
           -------------- + -------------------------
           3 (sin(x) - 1)             2          3
                                 sin(x)  + sin(x)  + 1
```

Changes:

- partfrac is now overloadable.

---

patchlevel – the patch number of the installed MuPAD library

patchlevel() returns the patch number of the currently installed MuPAD library.

Call(s):

- patchlevel()

Return Value: a nonnegative integer.

Related Functions: Pref::kernel, version

Details:

- patchlevel provides information about the patches installed in the local MuPAD setup. Patches (bug-fixes) to a release of the mathematical libraries are provided by Sciface Software or the MuPAD group. Whenever a new patch is installed, the patch level is increased by 1. The currently used library is determined by its version number (cf. the function version) together with its patch number.
- patchlevel is not related to the kernel version (Pref::kernel).
- Each new MuPAD version is initially released with patch number 0.
- To get information about new patches, please visit our web site at www.mupad.de.

Example 1. To query the version of the MuPAD library of your local installation, ask for its version number

```
>> version()

                              [2, 5, 0]
```

and for its patch number:

```
>> patchlevel()

                                  0
```

If the returned patch number is greater than zero, a patch was installed.

`pathname` – create a platform dependent path name

`pathname(dir, subdir, ...)` returns a relative path name valid on the used operating system.

Call(s):

- `pathname(dir, subdir, ..)`
- `pathname(`*Root* `, dir, subdir, ..)`

Parameters:

`dir, subdir, ..` — names of directories: character strings

Options:

Root — makes `pathname` generate an absolute path name

Return Value: a string.

Related Functions: `fclose`, `fileIO`, `finput`, `fopen`, `fprint`, `fread`, `ftextinput`, `LIBPATH`, `loadproc`, `package`, `print`, `protocol`, `read`, `READPATH`, `write`, `WRITEPATH`

Details:

- `pathname` is used to specify pathnames via MuPAD strings. Directories and subdirectories are concatenated in a suitable way creating a valid pathname for the currently used operating system. For example, this mechanism may be used to specify the location of library files independent of the platform.
- In order to create valid path names for the operating systems supported by MuPAD, the conventions holding for the corresponding operating system must be complied with. In particular, the names must not contain the characters "`/`" , "\" or "`:`". Compliance with these conventions is tested by `pathname`.
- Under Windows, `pathname` does not allow to specify a volume to become part of the path name. Names are always relative to the current volume.
- Examples:

call	result	platform
`pathname("lib", "linalg")`	`"lib/linalg/"`	UNIX/Linux
	`"lib\\linalg\\"`	Windows
	`":lib:linalg:"`	MacOS
`pathname(Root, "lib", "linalg")`	`"/lib/linalg/"`	UNIX/Linux
	`"\\lib\\linalg\\"`	Windows
	`"lib:linalg:"`	MacOS

- For an overview of all file related MuPAD functions, also try `?fileIO`.

Example 1. The following examples are created on a UNIX/Linux system:

```
>> pathname("lib", "linalg")

                    "lib/linalg/"

>> pathname(Root, "lib", "linalg") . "det.mu"

                 "/lib/linalg/det.mu"
```

`pdivide` – pseudo-division of polynomials

`pdivide(p, q)` computes the pseudo-division of the univariate polynomials `p` and `q`.

Call(s):

- `pdivide(p, q <, mode>)`
- `pdivide(f, g <, [x]> <, mode>)`

Parameters:

`p, q` — univariate polynomials of type `DOM_POLY`.
`f, g` — arithmetical expressions
`x` — an identifier or an indexed identifier. Multivariate expressions are regarded as univariate polynomials in the indeterminate `x`.

Options:

`mode` — either *Quo* or *Rem*. With *Quo*, only the pseudo-quotient is returned; with *Rem*, only the pseudo-remainder is returned.

Return Value: a polynomial, or a polynomial expression, or a sequence of an element of the coefficient ring of the input polynomials and two polynomials/polynomial expressions, or the value `FAIL`.

Overloadable by: `p, q, f, g`

Related Functions: `content`, `degree`, `divide`, `factor`, `gcd`, `gcdex`, `ground`, `lcoeff`, `multcoeffs`, `poly`

Details:

- `pdivide(p, q)` computes the pseudo-division of the univariate polynomials `p` and `q`. It returns the sequence `b, s, r`, where `b` = `lcoeff(q)^(degree(p) - degree(q) + 1)` is an element of the coefficient ring of the polynomials. The polynomials `s` (the pseudo-quotient) and `r` (the pseudo-remainder) satisfy $b\,p = s\,q + r$, $\texttt{degree}(p) = \texttt{degree}(s) + \texttt{degree}(q)$, $\texttt{degree}(r) < \texttt{degree}(q)$.

- The first two arguments can be either polynomials or arithmetical expressions.

  Polynomials must be of the same type, i.e., their variables and coefficient rings must be identical.

  Expressions are internally converted to polynomials (see the function `poly`). If no indeterminate `x` is specified, all symbolic variables in the expressions are regarded as indeterminates. `FAIL` is returned if more than one indeterminate is found. `FAIL` is also returned if the expressions cannot be converted to polynomials.

  The resulting polynomials have the same type as the first two arguments, i.e., they are either polynomials of type `DOM_POLY` or polynomial expressions.

- In contrast to `divide`, `pdivide` does not require that the coefficient ring of the polynomials implements a `"_divide"` slot: coefficients are not divided in this algorithm.

- `pdivide` is a function of the system kernel.

Example 1. This example shows the result of the pseudo-division of two polynomials:

```
>> p:= poly(x^3 + x + 1):  q:= poly(3*x^2 + x + 1):
   [b, s, r] := [pdivide(p, q)]

          [9, poly(3 x - 1, [x]), poly(7 x + 10, [x])]
```

The result satisfies the following equation:

```
>> multcoeffs(p, b) = s*q + r

              3                             3
     poly(9 x  + 9 x + 9, [x]) = poly(9 x  + 9 x + 9, [x])
```

Pseudo-quotients and pseudo-remainders can be computed separately:

```
>> pdivide(p, q, Quo), pdivide(p, q, Rem)

             poly(3 x - 1, [x]), poly(7 x + 10, [x])

>> delete p, q, b, s, r:
```

Example 2. The coefficient ring can be an arbitrary ring, e.g., the residue class ring of integers modulo 8:

```
>> pdivide(poly(x^3 + x + 1, IntMod(8)),
           poly(3*x^2 + x + 1, IntMod(8)))

 1, poly(3 x - 1, [x], IntMod(8)), poly(- x + 2, [x], IntMod(8))
```

Example 3. Here the input consists of multivariate polynomial expressions which are regarded as univariate polynomials in `x`:

```
>> pdivide(x^3 + x + y, a*x^2 + x + 1, [x])

             2             2
            a , a x - 1, a  y + x (a (a - 1) + 1) + 1
```

Example 4. The first argument cannot be converted to a polynomial. The return value is `FAIL`:

```
>> pdivide(1/x, x)

                                FAIL
```

`piecewise` – the domain of conditionally defined objects

`piecewise([condition1, object1], [condition2, object2], ...)` generates a conditionally defined object that equals `object1` if `condition1` is satisfied, `object2` if `condition2` is satisfied, etc.

Creating Elements:

- `piecewise([condition1, object1], [condition2, object2], ...)`

Parameters:

`condition1, condition2, ...`	—	Boolean constants or expressions representing logical formulas
`object1, object2, ...`	—	arbitrary objects

Side Effects: Properties of identifiers set by `assume` are taken into account.

Related Functions: `_case`, `_if`, `assume`, `bool`, `is`

Details:

- `piecewise` differs from the `if` and `case` branching statements in two ways. First, the property mechanism is used to decide the truth of the conditions. Hence the result depends on the properties of the identifiers that appear in the conditions. Second, `piecewise` treats conditions mathematically, while `if` and `case` evaluate them syntactically. Cf. example 2.

- A pair `[condition, object]` is called a *branch*. If `condition` is provably false, then the branch is discarded altogether. If `condition` is provably true, then `piecewise` returns `object`. If none of the conditions is provably true, an object of type `piecewise` is created containing all branches that have not been discarded.

  If all conditions are provably false, or if no branch is given, then `piecewise` returns `undefined`. Cf. example 1.

- The conditions need not be exhaustive, nor need they exclude each other. If you substitute values for the occurring parameters, it may happen that all conditions become false, but it may also happen that more than one condition becomes true.

- If several conditions are simultaneously true, `piecewise` returns the first object defined under a condition that is *recognized* to be true. The user has to ensure that the objects corresponding to the true conditions all have the same mathematical meaning. You cannot rely on the system to recognize the first mathematically true condition as true.

- Whenever an object of type `piecewise` is evaluated, the truth of the conditions is checked again for the current values and the current properties of the identifiers involved. This may be used to simplify the result of a computation under various different assumptions.

- Conditionally defined objects may be nested: both conditions and objects may be conditionally defined themselves. `piecewise` automatically denests ("flattens") such objects. For example, "if A then (if B then C)" becomes "if A and B then C". Cf. example 7.

- Arithmetical and set-theoretic operations work for conditionally defined objects, provided these operations are defined for all objects contained in the branches. If `f` is such an operation and `p1, p2, ...` are conditionally defined objects, then `f(p1, p2, ...)` is the conditionally defined object consisting of all branches of the form `[condition1 and condition2 and ..., f(object1, object2, ...)]`, where `[condition1, object1]` is a branch of `p1`, `[condition2, object2]` is a branch of `p2`, etc. This can also be understood as follows: applying `f` commutes with any assignment to free parameters in the conditions.

Conditionally defined objects can also be mixed with other objects in such operations: If, e.g., `p1` is not a conditionally defined object, it is handled like a conditionally defined object with the only branch `[TRUE, p1]`.

Cf. examples 3 and 6.

- In particular, the previous remark holds for unary operators and functions with one argument: if called with a conditionally defined object as argument, they are mapped to the objects in each branch. Cf. example 5.

Mathematical Methods

Method `_in`: membership with `piecewise` on the left hand side

`_in(`*`piecewise`*` p, `*`set`*` S)`

- This method returns a logical formula that is equivalent to "`p` is an element of `S`".
- This method overloads `_in`.

Method `contains`: apply the function `contains` to the objects in all branches

`contains(`*`piecewise`*` p, `*`any`*` a)`

- This method applies the function `contains` with second argument `a` to the objects in all branches of `p`. The result is in general again a conditionally defined object.
- This method overloads the function `contains`. The objects in all branches must be valid first arguments for `contains`.

Method `diff`: (partial) differentiation

`diff(`*`piecewise`*` p <, `*`identifier`*` x, ...>)`

- This method differentiates the objects in all branches of `p` with respect to the given variables, starting with the leftmost one.
- If no variables are given, `p` is returned.
- This method overloads `diff`.

Method `discont`: determine the discontinuities of a piecewise defined function

`discont(`*piecewise* `p,` *identifier* `x <,` *domain* `F>)`

- This method returns a superset of the discontinuities of `p` regarded as a function depending on `x`, namely, the union of the results of applying `discont` to the objects in all branches of `p`, plus the boundary points of the conditions of `p` with respect to `x`.
- The objects in all branches of `p` must be arithmetical expressions.
- This method overloads `discont`.
- The optional third parameter has the same meaning as for the function `discont`.

`discont(`*piecewise* `p, x=a..b <,` *domain* `F>)`

- As for the function `discont`, only discontinuities in the given interval are returned.

Method `piecewise::disregardPoints`: heuristic for simplifying conditions

`piecewise::disregardPoints(`*piecewise* `p)`

- Apply the heuristic "consider equalities to be false" to conditions that are recognized neither as true nor as false. Since the set of zeroes of an equation usually has Lebesgue measure zero, this heuristic tends to produce a simpler case distinction that is equivalent to the original one for almost all values of the parameters. Cf. example 6.

Method `expand`: apply the function `expand` to the objects in all branches

`expand(`*piecewise* `p)`

- This method overloads `expand`.

Method `factor`: apply the function `factor` to the objects in all branches

`factor(`*piecewise* `p)`

- This method overloads `factor`.

Method `piecewise::getElement`: get any element of a conditionally defined set

piecewise::getElement(*piecewise* p)

- This method returns an element that is common to the objects in all branches of `p`. All such objects must represent sets.
- The result is `FAIL` if no such common element can be found.
- This method overloads the function `solvelib::getElement`.

Method `has`: test for the existence of a subobject

has(*piecewise* p, *any* a)

- This method tests whether `a` appears syntactically somewhere in the conditions or the objects of `p`; it returns `TRUE` if this is the case, and `FALSE` otherwise.
- This method overloads `has`.

Method `int`: definite and indefinite integration of a piecewise defined function

int(*piecewise* p, *identifier* x <, *range* r>)

- If no range is given, this method computes the indefinite integral of `p`, where `p` is regarded as a piecewise defined function of `x`. It applies the function `int` to the objects in all branches of `p`.
- If a range `a..b` is given, this method computes the definite integral of `p` when `x` runs through that range.
- This method overloads `int`.

Method `piecewise::invlaplace`: apply the function `transform::invlaplace` to the objects in all branches

piecewise::invlaplace(*piecewise* p, *identfier* x, *identifier* t)

- This method overloads the function `transform::invlaplace`.

Method `piecewise::isFinite`: test whether a piecewise defined set is finite

`piecewise::isFinite(`*`piecewise`*` p)`

- This method returns `TRUE` if the objects in all branches of `p` are finite sets, and it returns `FALSE` if the objects in all branches of `p` are infinite sets. Otherwise, it returns `UNKNOWN`.
- This method overloads `solvelib::isFinite`.

Method `piecewise::laplace`: apply the function `transform::laplace` to the objects in all branches

`piecewise::laplace(`*`piecewise`*` p, `*`identifier`*` x, `*`identifier`*` t)`

- This method overloads the function `transform::laplace`.

Method `normal`: apply the function `normal` to the objects in all branches

`normal(`*`piecewise`*` p)`

- This method overloads `normal`.

Method `partfrac`: apply the function `partfrac` to the objects in all branches

`partfrac(`*`piecewise`*` p)`

- This method overloads `partfrac`.

Method `piecewise::restrict`: impose an additional condition

`piecewise::restrict(`*`any`*` p, `*`condition`*` C)`

- If `p` is not a conditionally defined object, this method creates the conditionally defined object with a single branch `[C, p]`. If `p` is conditionally defined, each condition `cond` in `p` is replaced by `cond and C`.

Method `piecewise::set2expr`: membership with `piecewise` on the right hand side

`piecewise::set2expr(`*`piecewise`*` p, `*`identifier`*` x)`

- This method returns a logical formula with free parameter `x` that is equivalent to "`x` is an element of `p`".
- The objects in all branches of `p` must represent sets.
- This method overloads the system function `_in`.

Method `simplify`: simplify a conditionally defined object

`simplify(`*`piecewise`*` p)`

- This method performs the following simplifications:

  - First, `simplify` is applied to the objects in all branches.
  - Branches defining the same object are collected.
  - If the condition of some branch implies that a free parameter is constant, the parameter is replaced by that constant in the object of that branch.

  Cf. example 7.
- This method overloads `simplify`.

Method `solve`: solve a conditionally defined equation or inequality

`solve(`*`piecewise`*` p, `*`identifier`*` x <, option1, option2, ...>)`

- This method solves `p` for the variable `x`. The objects in all branches of `p` must be either equations, inequalities, or arithmetical expressions; each arithmetical expression `e` is replaced by an equation `e = 0`.
- For each branch `[condition, object]` of `p`, with `object` being an equation or inequality, the method determines the set of all values `x` such that both `condition` and `object` become true mathematically, and returns the union of all obtained sets. The return value may be a conditionally defined set.
- This method overloads the function `solve`. See the corresponding help page for a description of the available options and an overview of the types of sets that may be returned.

Method `piecewise::solveConditions`: isolate a given identifier in all conditions

`piecewise::solveConditions(`*piecewise* `p,` *identifier* `x)`

- This method rewrites each condition of `p` containing `x` in the form "`x in S`" for some appropriate set `S`.

Method `piecewise::Union`: union of a system of sets

`piecewise::Union(`*piecewise* `p,` *identifier* `x,` *set* `indexset)`

- This method returns the set of all elements of elements of `p`, where `p` is regarded as a system of sets parameterized by `x` and `x` runs through all elements of `indexset`.
- The objects in all branches of `p` must represent sets.
- For each branch `[condition, object]` of `p`, this method does the following. It substitutes for `x` in `object` all those values from `indexset` satisfying `condition` and takes the union over all obtained sets. Then it returns the union over the resulting sets for all branches.
- This method overloads the function `solvelib::Union`.

Access Methods

Method `_concat`: merge piecewise objects

`_concat(`*piecewise* `p, ...)`

- This method returns a conditionally defined object comprising the branches of all arguments.
- This method overloads `_concat`.

Method `piecewise::branch`: nth branch

`piecewise::branch(`*piecewise* `p,` *positive integer* `n)`

- This method returns the `n`th branch of `p` as a list.

Method `piecewise::numberOfBranches`: number of branches

`piecewise::numberOfBranches(`*piecewise* `p)`

- This method returns the number of branches of `p`.

Method `piecewise::condition`: the condition in a specific branch

`piecewise::condition(`*piecewise* `p,` *positive integer* `i)`

- This method returns the condition of the `i`th branch of `p`. Cf. example 4.

Method `piecewise::expression`: the object in a specific branch

`piecewise::expression(`*piecewise* `p,` *positive integer* `i)`

- This method returns the object of the `i`th branch of `p`. Cf. example 4.

Method `piecewise::insert`: insert a branch at a given position

`piecewise::insert(`*piecewise* `p,` *branch* `b,` *positive integer* `i)`

- This method returns `p` with the branch `b` inserted at position `i`.
- `b` can either be a branch extracted from another conditionally defined object using `extop`, or a list `[condition, object]`.
- The integer `i` must not exceed the number of branches of `p` plus one.
- Cf. example 4.

Method `piecewise::extmap`: apply a function to the objects in all branches

`piecewise::extmap(`*piecewise* `p,` *any* `f` $<$`,` *any* `a, ...`$>$`)`

- For each branch `[condition, object]` of `p`, `object` is replaced by `f(object` $<$`, a, ...`$>$`)`.

Method `piecewise::mapConditions`: apply a function to the conditions in all branches

`piecewise::mapConditions(`*piecewise* `p,` *any* `f` $<$`,` *any* `a, ...`$>$`)`

- For each branch `[condition, object]` of `p`, `condition` is replaced by `f(condition` $<$`, a, ...`$>$`)`.

Method `map`: apply the function `map` to the objects in all branches

`map(`*any* `p,` *any* `f <,` *any* `a, ...>)`

- For each branch `[condition, object]` of `p`, `object` is replaced by `map(object, f <, a, ...>)`.
- This method overloads `map`.

Method `op`: apply the function `op` to the objects in all branches

`op(`*piecewise* `p <, furtherargs, ...>)`

- For each branch `[condition, object]` of `p`, `object` is replaced by `op(object<,furtherargs, ...>)`.

Method `piecewise::remove`: remove a branch

`piecewise::remove(`*piecewise* `p,` *positive integer* `i)`

- This method returns a conditionally defined object obtained from `p` by deleting the `i`th branch. Cf. example 4.

Method `piecewise::splitBranch`: split a branch into two branches

`piecewise::splitBranch(`*piecewise* `p,` *positive integer* `i,` *condition* `newcondition)`

- This method returns a conditionally defined object obtained from `p` by splitting the `i`th branch into two branches. Let `[condition, object]` be the `i`th branch; then the new branches are `[condition and newcondition, object]` and `[condition and not newcondition, object]`.

Method `piecewise::selectConditions`: select branches depending on their condition

`piecewise::selectConditions(`*piecewise* `p,` *any* `f <,` *any* `a, ...>)`

- This method works like the function `select` with the selection criterion given by `f` applied to the conditions of `p`. It returns the piecewise object derived from `p` by removing every branch `[condition, object]` for which `f(condition <, a, ...>)` does not yield `TRUE`.
- For every condition in `p`, `f(condition <, a, ...>)` must return a Boolean constant.
- If none of the conditions satisfies the selection criterion, `undefined` is returned.

Method `piecewise::splitConditions`: split branches depending on conditions

`piecewise::splitConditions(`*piecewise* `p,` *any* `f <,` *any* `a, ...>)`

- This method works like the function `split` with the splitting criterion given by `f` applied to the conditions of `p`. It returns a list of three conditionally defined objects, comprising those branches `[condition, object]` of `p` for which `f(condition <, a, ...>)` yields `TRUE`, `FALSE`, and `UNKNOWN`, respectively.

  If, for some of the three Boolean values, no branch yields that value, then the returned list contains `undefined` instead of a conditionally defined object with zero branches at the corresponding position.
- For every condition in `p`, `f(condition <, a, ...>)` must return a Boolean constant.
- Cf. example 4.

Method `subs`: substitution

`subs(`*piecewise* `p,` *substitution* `s, ...)`

- This method performs the substitution(s) `s` in both the conditions and the objects of `p`.
- This method overloads the function `subs`. The calling syntax is identical to that function; cf. the corresponding help page for a description of the various types that are allowed for `s`.

Method `zip`: apply a binary operation pointwise

`zip(`*any* `p1,` *any* `p2,` *any* `f)`

- If both `p1` and `p2` are conditionally defined objects, then this method returns the conditionally defined object comprising all branches of the form `[condition1 and condition2, f(object1, object2)]`, where `[condition1, object1]` is a branch of `p1` and `[condition2, object2]` is a branch of `p2`.
- If we regard conditionally defined objects as functions from the set A of parameter values to a set B of objects, this method implements the canonical extension of the binary operation f on B to the binary operation g on the set B^A of all functions from A to B via $(g(p1, p2))(a) = f(p1(a), p2(a))$ for all a in A.
- If only one of the first two arguments—`p1`, say—is of type `piecewise`, then each branch `[condition, object]` of `p1` is replaced by `[condition, f(object, p2)]`.

- If neither `p1` nor `p2` are of type `piecewise`, then `piecewise::zip(p1, p2, f)` returns `f(p1, p2)`.
- This method overloads `zip`.

Example 1. We define `f` as the characteristic function of the interval $[0, 1]$:

```
>> f := x -> piecewise([x < 0 or x > 1, 0], [x >= 0 or x <= 1, 1])

   x -> piecewise([x < 0 or 1 < x, 0], [0 <= x or x <= 1, 1])
```

None of the conditions can be evaluated to `TRUE` or `FALSE`, unless more is known about the variable `x`. When we evaluate `f` at some point, the conditions are checked again:

```
>> f(0), f(2), f(I)

                        1, 0, undefined
```

Example 2. `piecewise` performs a case analysis using the property mechanism. It checks whether the given conditions are *mathematically* true or false; it may also decide that not enough information is available. In the following example, it cannot be decided whether `a` is zero as long as no assumptions on `a` have been made:

```
>> delete a:
   p := piecewise([a = 0, 0], [a <> 0, 1/a])

                  /             1           \
         piecewise| 0 if a = 0, - if a <> 0 |
                  \             a           /
```

In contrast, `if`-statements evaluate the conditions syntactically: `a=0` is *technically* false since the identifier `a` and the integer `0` are different objects:

```
>> if a = 0 then 0 else 1/a end

                              1
                              -
                              a
```

Moreover, `piecewise` takes properties of identifiers into account:

```
>> assume(a = 0):
   p;
   delete a, p:

                              0
```

Example 3. Conditionally defined objects can be created by rewriting special functions:

```
>> f := rewrite(sign(x), piecewise)

          /
 piecewise| 1 if 0 < x, -1 if x < 0, 0 if x = 0,
          |
          \

              x                               \
    -------------------- if x in C_ minus R_ |
          2        2 1/2                      |
    (Im(x)  + Re(x) )                         /
```

In contrast to MuPAD, most people like to regard `sign` as a function defined for real numbers only. You might therefore want to restrict the domain of `f`:

```
>> f := piecewise::restrict(f, x in R_)

         piecewise(1 if 0 < x, -1 if x < 0, 0 if x = 0)
```

Conditionally defined arithmetical expressions allow roughly the same operations as ordinary arithmetical expressions. The result of an arithmetical operation is only defined at those points where all of the arguments are defined:

```
>> f + piecewise([x < 2, 5])

     piecewise(6 if 0 < x and x < 2, 4 if x < 0, 5 if x = 0)
```

Example 4. There are several methods for extracting branches, conditions, and objects. Consider the following conditionally defined object:

```
>> f := piecewise([x > 0, 1], [x < -3, x^2])

                                        2
               piecewise(1 if 0 < x, x  if x < -3)
```

You can extract a specific condition or object:

```
>> piecewise::condition(f, 1), piecewise::expression(f, 2)

                                      2
                             0 < x, x
```

The function `extop` extracts whole branches:

```
>> extop(f, 1)
```

```
                    1  if 0 < x
```

You can form another piecewise defined object out of those branches for which the condition satisfies a given selection criterion, or split the input into two piecewise defined objects, as the system functions `select` and `split` do it for lists:

```
>> piecewise::selectConditions(f, has, 0)

                      piecewise(1 if 0 < x)

>> piecewise::splitConditions(f, has, 0)

                                             2
   [piecewise(1 if 0 < x), piecewise(x  if x < -3), undefined]
```

You can also create a copy of `f` with some branches added or removed:

```
>> piecewise::remove(f, 1)

                                 2
                      piecewise(x  if x < -3)

>> piecewise::insert(f, [x > -3 and x < 0, sin(x)], 2)

                                                          2
 piecewise(1 if 0 < x, sin(x) if x < 0 and -3 < x, x  if x < -3)
```

Example 5. Most unary functions are overloaded for `piecewise` by mapping them to the objects in all branches of the input. This can also be achieved using `piecewise::extmap`:

```
>> f := piecewise([x >= 0, arcsin(x)], [x < 0, arccos(x)]):
   sin(f)

                                    2      1/2
         piecewise(x if 0 <= x, (- x  + 1)    if x < 0)

>> piecewise::extmap(f, sin)

                                    2      1/2
         piecewise(x if 0 <= x, (- x  + 1)    if x < 0)
```

Example 6. Sets may also be conditionally defined. Such sets are sometimes returned by `solve`:

```
>> S := solve(a*x = 0, x)

              piecewise(C_ if a = 0, {0} if a <> 0)
```

The usual set-theoretic operations work for such sets:

```
>> S intersect Dom::Interval(3, 5)

            piecewise(]3, 5[ if a = 0, {} if a <> 0)
```

Sometimes it is interesting to exclude the "rare cases" which only cover a small set of parameter values:

```
>> piecewise::disregardPoints(S)

                                {0}
```

Example 7. Consider the following case distinction:

```
>> p1 := piecewise([a > 0, a^2], [a <= 0, -a^2]):
   p2 := piecewise([b > 0, a + b], [b = 0, p1 + b], [b < 0, a + b])

 piecewise(a + b if b <> 0 and b in R_,

         2                           2
    b + a  if 0 < a and b = 0, b - a  if b = 0 and a <= 0)
```

Note that the system has moved the case analysis done in `p1` to the top level automatically. However, some simplifications are still possible: the branches `b>0` and `b<0` can be collected, and in the case `b=0` the identifier `b` may be replaced by the value `0`:

```
>> simplify(p2)

                                                2
 piecewise(a + b if b <> 0 and b in R_, a  if 0 < a and b = 0,

       2
    - a  if b = 0 and a <= 0)
```

Background:

- The operands of a conditionally defined object, i.e., the branches, are pairs consisting of a condition and the object valid under that condition. They are of a special data type `stdlib::branch`.
- Methods overloading system functions always assume that they have been called via overloading, and that there is some conditionally defined object among their arguments. All other methods do not assume that one of their arguments is of type `piecewise`. This simplifies the use of `piecewise`: it is always allowed to enter `p:=piecewise(...)` and to call some method of `piecewise` with `p` as argument. You need not care about the special case where `p` is not of type `piecewise` because some condition in its definition is true or all conditions are false.

Changes:

- The method `map` now applies the function `map` to all branches; this used to be `piecewise::mapMap` in version 2.0. The method `piecewise::extmap` must be used to map a given function to all branches.
- The method `op` now applies the function `op` to all branches; this used to be `piecewise::mapOp` in version 2.0. The function `extop` must be used to extract branches.
- The method `discont` now accepts a second argument of the form `x=a..b`, too.
- New methods `branch`, `splitBranch`, and `numberOfBranches` have been added.
- New methods `laplace`, `invlaplace`, `partfrac`, and `factor` have been added.

`plot` – display graphical objects on the screen

`plot(scene)` displays a graphical scene on the screen.

`plot(object1, object2, ...)` displays the graphical objects `object1`, `object2` etc. on the screen.

Call(s):

- `plot(scene)`
- `plot(object1 <, object2, ...> <, option1, option2, ...>)`

Parameters:

`scene`	—	a graphical scene: an object of domain type `plot::Scene`
`object1, object2, ...`	—	2D or 3D graphical objects
`option1, option2, ...`	—	scene options of the form `OptionName = value`

Overloadable by: `object1`

Related Domains: `plot::Scene`

Related Functions: `plot2d`, `plot3d`

Details:

- Graphical scenes may be created by `plot::Scene`. See the corresponding help page for details.
- The parameters `object1`, `object2` etc. must be graphical objects generated by routines of the library `plot`. Graphical primitives include function graphs (of domain type `plot::Function2d` and `plot::Function3d`), points and polygons (of domain type `plot::Point` and `plot::Polygon`, respectively), and surfaces (of domain type `plot::Surface3d`). Cf. example 1.
- High level functions of the `plot` library such as `plot::vectorfield`, `plot::ode`, or `plot::implicit` return more complex graphical objects that can also be rendered via the function `plot`. Cf. example 2.
- Scene options `option1`, `option2` etc. are specified by equations `OptionName = value`. Please refer to the help page of `plot::Scene` for a table of all admissible plot options.
- The graphical objects `object1`, `object2` etc. must have the same dimension. A mix of two- and three-dimensional primitives in a single scene is not supported! NOTE

Example 1. The following calls return objects representing the graphs of the sine and the cosine function on the interval $[0, 2\pi]$:

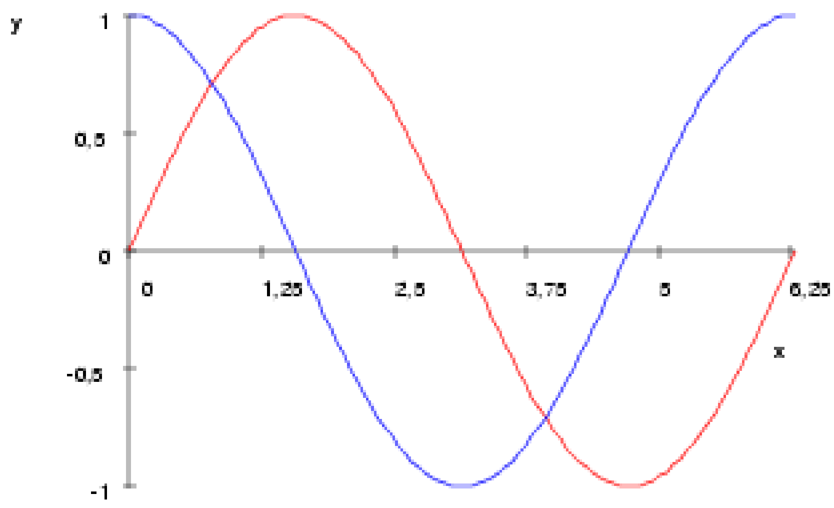
```
>> f1 := plot::Function2d(sin(x), x = 0..2*PI);
   f2 := plot::Function2d(cos(x), x = 0..2*PI, Color = RGB::Blue)

              plot::Function2d(sin(x), x = 0..2 PI)

              plot::Function2d(cos(x), x = 0..2 PI)
```

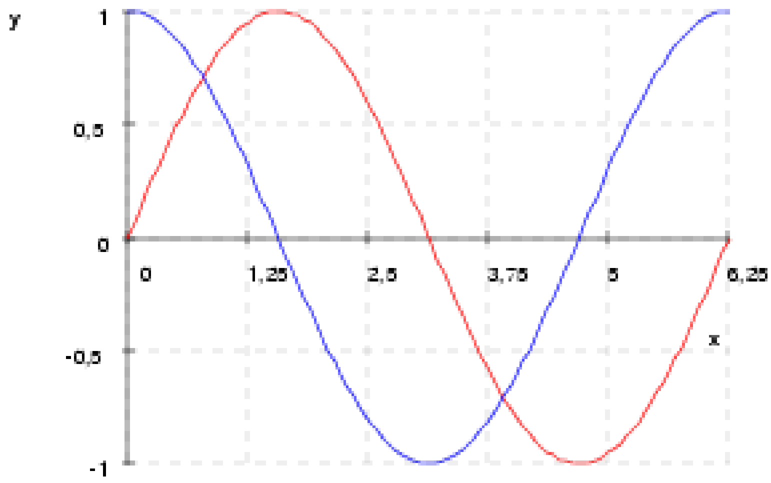
The following call renders these graphs:

```
>> plot(f1, f2)
```



This call uses the default values of the scene options as documented on the help page of `plot::Scene`. Scene options may be passed as additional parameters to `plot`. For example, to draw grid lines in the background of the previous plot, we enter:

```
>> plot(f1, f2, GridLines = Automatic)
```



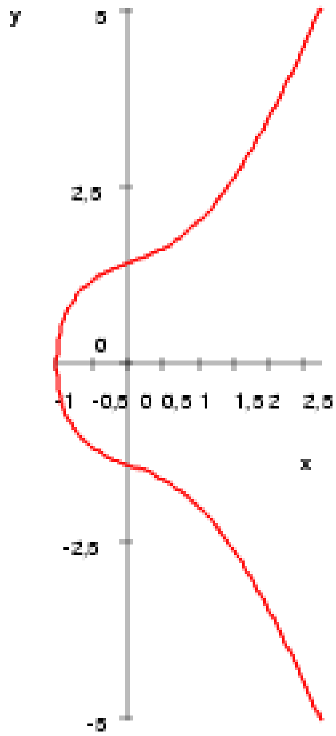
See `plotOptions2d` for details on the `GridLines` option.

```
>> delete f1, f2:
```

Example 2. The `plot` library contains various routines for creating more complex graphical objects such as vectorfields, solution curves of ordinary differential equations, and implicitly defined curves.

For example, to plot the implicitly defined curve $x^2 + x + 2 = y^2$ with x, y from the interval $[-5, 5]$, we use the function `plot::implicit`:

```
>> plot(
        plot::implicit(
           x^3 + x + 2 - y^2, x = -5..5, y = -5..5
        ),
        Scaling = Constrained
       )
```



Here we used the `Scaling` option to guarantee an aspect ratio 1:1 between the x and y coordinates independent of the window size (see `plotOptions2d` for details).

Background:

- Technically, `plot` is not a function but a domain representing the library `plot`. Thus, when calling `plot(...)`, the method `plot::new(...)` is called.

  The method `"new"` works as follows: If the parameter `scene` is given, the method `"getPlotdata"` of the domain `plot::Scene` is called. It returns the graphical scene in a `plot2d` conforming syntax (or `plot3d` if the scene is three-dimensional). Then the result is passed to the function `plot2d` or `plot3d`, respectively.

  If the graphical objects `object1`, `object2` etc. are given as parameters, the method `plot::new` first creates a scene of domain type `plot::Scene` consisting of these objects. Then it proceeds as described above.

`plot2d` – **2D plots**

`plot2d(object1, object2, ...)` generates a 2D plot of graphical objects such as parametrized curves, points, and polygons.

Call(s):

- `plot2d(`<`SceneOptions,`> `object1, object2, ...)`

Parameters:

`object1, object2, ...` — graphical objects as described below

Options:

`SceneOptions` — a sequence of scene options. These determine the general appearance of the graphical scene. See `?plotOptions2d` for details.

Return Value: MuPAD's graphics tool is called to render the graphical scene, and the `null()` object is returned to the MuPAD session.

Related Functions: `plot`, `plotfunc2d`, `plot3d`, `plotfunc3d`

Details:

- `plot2d` is a low level interface to create 2D plots from graphical primitives. For graphs of functions, the specialized routines `plotfunc2d` and `plot::Function2d` are more convenient. For graphical scenes built from primitives, we recommend to use the `plot` library, which provides various primitives and tools. In most cases, the user will find it more convenient to use the `plot` library rather than `plot2d`.
- There are two types of graphical objects that are accepted by `plot2d`: i) lists of graphical primitives (points and polygons) and ii) parametrized curves.
- i) *Lists of graphical primitives* are objects of the following form:

  `[`*`Mode`*` = `*`List`*` , [primitive1, primitive2, ...] `<`, Options`>`]`

  The available primitives are points, polygons and filled polygons generated by the MuPAD functions `point` and `polygon`, respectively. You can use such primitives to build more complicated graphical objects.

  Options are specified by equations `OptionName = value`. The following table gives an overview of the available options:

OptionName	admissible values	default value
Color	[*Flat*], [*Flat*, [r,g,b]], [*Height*], [*Height*, [r,g,b], [R,G,B]], [*Function*, f]	[*Height*]
LineStyle	*SolidLines*, *DashedLines*	*SolidLines*
LineWidth	positive integers	1
PointStyle	*Circles*, *FilledCircles*, *FilledSquares*, *Squares*	*FilledSquares*
PointWidth	positive integers	30
Title	strings	""
TitlePosition	[x, y]	

See the description below for further details on each option.

ii) *Parametric curves* are given by a parametrization $u \mapsto [x(u), y(u)]$ with expressions $x(u)$, $y(u)$ defining the x, y-coordinates as functions of a curve parameter u. In `plot2d`, a curve is defined by an object of the following form:

[*Mode* = *Curve*, [x(u), y(u)], u = [umin, umax] <, Options>]

The parametrization `x(u)`, `y(u)` consists of arithmetical expressions in one indeterminate `u` (an identifier). They must not contain any other symbolic parameters that cannot be converted to real floating point numbers. The range of the curve parameter `u` is given by the real numbers or numerical expressions `umin` and `umax`.

If the parametrization is given by user-defined functions that accept only numerical values, premature evaluation can be avoided using `hold(x)(u)`, `hold(y)(u)` with the symbolic curve parameter `u`.

Options are specified by equations `OptionName = value`. All options for a list of primitives can be used. For curves, the following additional options are available:

OptionName	admissible values	default value
Grid	[n]	[100]
Smoothness	[n]	[0]
Style	[*Points*], [*Lines*], [*LinesPoints*], [*Impulses*]	[*Lines*]

See the description below for further details on each option.

The graph of a function $f(x)$ can be plotted as a parametrized curve

[*Mode* = Curve, [x, f(x)], x = [xmin, xmax] <, Options>]

However, it is more convenient to use `plotfunc2d` or `plot::Function2d` to plot or generate function graphs. Furthermore, in contrast to `plot2d`, the latter handle functions with singularities.

- MuPAD graphics can be saved in a variety of graphical formats. In a `plot2d` command, the *PlotDevice* scene option allows to specify the conversion into the two MuPAD specific formats '*Ascii*' and '*Binary*'. See the help page `plotOptions2d` for details.

  For graphical standard formats such as *Postscript*, *JPEG*, *TIFF* etc., no direct conversion is available by a plot command inside a MuPAD session. Instead, conversion has to be requested interactively via the graphical interface of the rendering tool VCam. In a MuPAD Pro notebook, double click on the graphics to activate this interface. Using the menu item "Edit/Save Graphics ..", you can choose the desired format in the "Export Graphics" dialog box.

Option <`Color = value`>**:**

- This option determines the color of the object. Admissible values are [*Flat*], [*Flat*, [r,g,b]], [*Height*], [*Height*, [r,g,b], [R,G,B]] and [*Function*, f]. The default is *Color* = [*Height*].

  - With *Color* = [*Flat*], the object is displayed with a flat color. The actual color is chosen automatically.

  - With *Color* = [*Flat*, [r, g, b]], the object is displayed with a flat color. The values `r`, `g`, `b` represent the red, green and blue contributions according to the RGB color model. They must be real numbers between `0` and `1`. Pre-defined colors are provided by MuPAD's `RGB` data structure.

  - With *Color* = [*Height*], the color varies with the y-coordinate. The actual colors are chosen automatically.

  - With *Color* = [*Height*, [r, g, b], [R, G, B]], the color varies with the y-coordinate. The parts of the object with small values of y are displayed with the color `[r, g, b]`, parts with large values of y are displayed with the color `[R, G, B]`. Interpolated color values are used in between.

  - With *Color* = [*Function*, f], users may implement their own coloring scheme. The parameter `f` must be a MuPAD procedure returning a color as a list `[r, g, b]`.

    – Inside a curve object, the function `f` must accept three parameters:

    ```
    f := proc(x, y, u) begin ...; return([r, g, b]) end:
    ```

    During the numerical evaluation of the plot, this function is called with the arguments `(x(u), y(u), u)`, where `u` is the curve parameter and `x(u)`, `y(u)` are the corresponding coordinates.

– Inside a list of primitives, the function `f` must accept two parameters:

  `f := proc(x, y) begin ...; return([r, g, b]) end:`

  During the numerical evaluation of the plot this function is called with arguments `(x, y)` from the viewing range of the object.

  Note that polygons are always displayed with a flat color.

If the color function `f` is created inside a procedure, using local variables of this procedure, then this procedure must use `option escape`.

Option <`Grid = [n]`>**:**

- This option determines the number of sample points of the curve. The graphics uses linear interpolation between adjacent sample points. The integer `n` must be larger than 1. The default is *Grid* = `[100]`. Large values of `n` generate a smooth curve. Alternatively, the *Smoothness* parameter can be increased.

Option <`LineStyle = value`>**:**

- This option determines the style in which lines are displayed. Admissible values are *SolidLines* and *DashedLines*; the default is *LineStyle* = *SolidLines*.

Option <`LineWidth = n`>**:**

- This option sets the width of the lines belonging to the object. Admissible values for `n` are nonnegative integers; the default is *LineWidth* = 1.

Option <`PointStyle = value`>**:**

- This option sets the style in which point objects are displayed. Admissible values are *Circles*, *Squares*, *FilledCircles*, and *FilledSquares*. The default is *PointStyle* = *FilledSquares*.

Option <`PointWidth = n`>**:**

- This option sets the size of point objects. Admissible values for `n` are positive integers; the default is *`PointWidth`* = 30.

Option <`Smoothness = [n]`>**:**

- This option determines the number of interpolation points between the sample points determined by the *`Grid`* option. Admissible values for `n` are integers between 0 and 20; the default is *`Smoothness`* = `[0]`. Lines are depicted as linear segments connecting these interpolation points. Consequently, large values of `n` produce smooth lines.

Option <`Style = value`>**:**

- This option sets the style in which curves are displayed. Admissible values are `[`*`Points`*`]`, `[`*`Lines`*`]`, `[`*`LinesPoints`*`]` and `[`*`Impulses`*`]`. The default is *`Style`* = `[`*`Lines`*`]`.

  - With *`Style`* = `[`*`Points`*`]`, only the sample points determined by the *`Grid`* option are displayed.
  - With *`Style`* = `[`*`Lines`*`]`, the curve is displayed as a collection of line segments connecting the sample points.
  - With *`Style`* = `[`*`LinesPoints`*`]`, both the sample points as well as the connecting line segments are displayed.
  - With *`Style`* = `[`*`Impulses`*`]`, the curve is displayed like a "histogram": vertical lines from the bottom of the scene to the sample points are drawn.

Option <`Title = TitleString` >**:**

- This option adds the text given by the string `TitleString` to the object. The default is the empty string *`Title`* = `""`, i.e., no title.

Option <TitlePosition = [x, y]>:

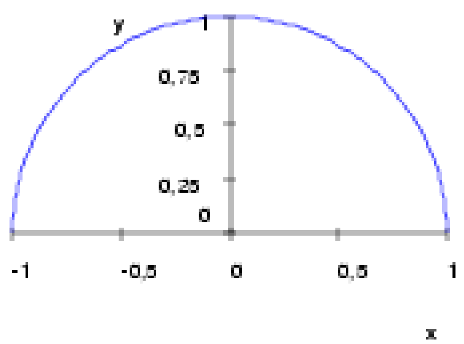
- This option determines the position of the object title. The parameters `x`, `y` must be numerical values between 0 and 10. The position `[0, 0]` denotes the upper left corner of the scene, the position `[10, 10]` denotes the lower right corner.

  Note that the specified positions are relative to the entire scene. Consequently, if titles are specified for several objects, their positions should differ to avoid overlap.

- Object titles can be moved interactively with the mouse to any appropriate position inside the scene.

Example 1. We plot a semi-circle of radius 1, parametrized by the polar angle u. The scene option *Scaling* = *Constrained* ensures that the circle is not deformed to an ellipse:

```
>> plot2d(Scaling = Constrained, Labeling = TRUE,
          [Mode = Curve, [cos(u), sin(u)], u = [0, PI]])
```



Example 2. We define two point primitives, a line primitive and a filled polygon:

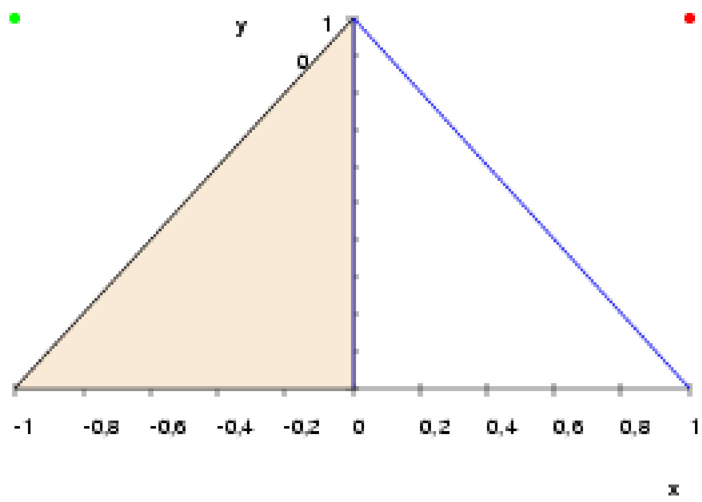
```
>> point1 := point(1, 1, Color = RGB::Red):
   point2 := point(-1, 1, Color = RGB::Green):
   line := polygon(point(1, 0), point(0, 1), point(0, 0),
                   Color = RGB::Blue):
   triangle := polygon(point(0, 0), point(0, 1), point(-1, 0),
                       Closed = TRUE, Filled = TRUE,
                       Color = RGB::Antique):
```

These are combined to a graphical object:

```
>> object := [Mode = List, [point1, point2, line, triangle]]:
```

Finally, this object is plotted:

```
>> plot2d(BackGround = RGB::White, PointWidth = 50,
          PointStyle = FilledCircles, object)
```

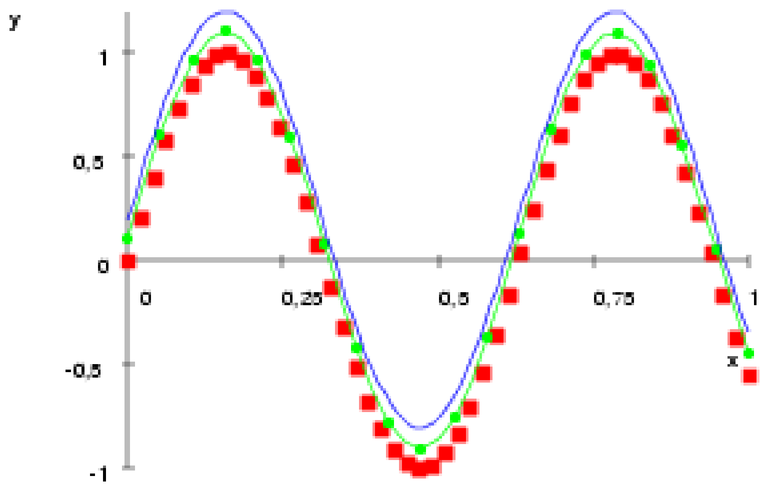



```
>> delete point1, point2, line, triangle, object:
```

Example 3. The graph of the sine function is diplayed using different styles:

```
>> plot2d(BackGround = RGB::White, ForeGround = RGB::Black,
          Labeling = TRUE, PointWidth = 50,
          [Mode = Curve, [x, sin(10*x)], x = [0, 1],
           Color = [Flat, RGB::Red], Grid = [50], Smoothness = [0],
           PointStyle = FilledSquares, Style = [Points]
```

```
],
[Mode = Curve, [x, 0.1 + sin(10*x)], x = [0, 1],
 Color = [Flat, RGB::Green],
 Grid = [20], Smoothness = [1],
 PointStyle = FilledCircles, Style = [LinesPoints]
],
[Mode = Curve, [x, 0.2 + sin(10*x)], x = [0, 1],
 Color = [Flat, RGB::Blue], Grid = [100], Style = [Lines]
])
```

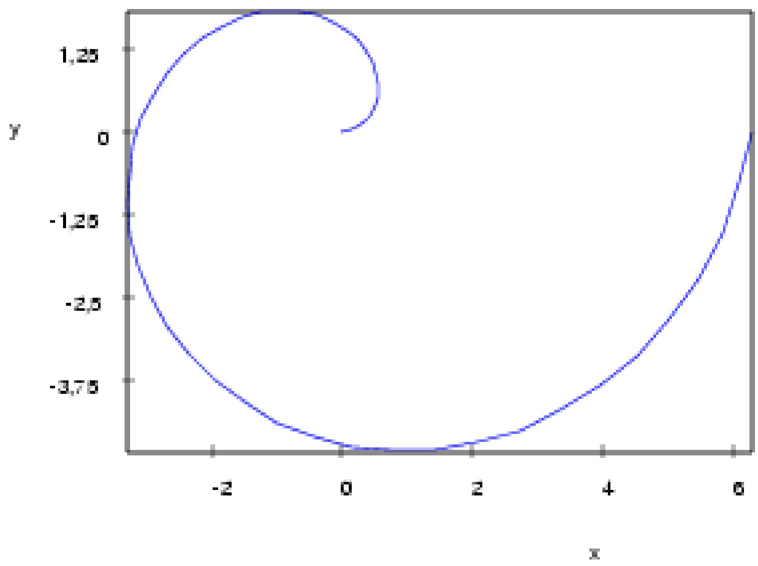


Example 4. We demonstrate the *ViewingBox* option.

```
>> spiral := [Mode = Curve, [u*cos(u), u*sin(u)], u = [0, 2*PI],
              Grid = [50]]:
```

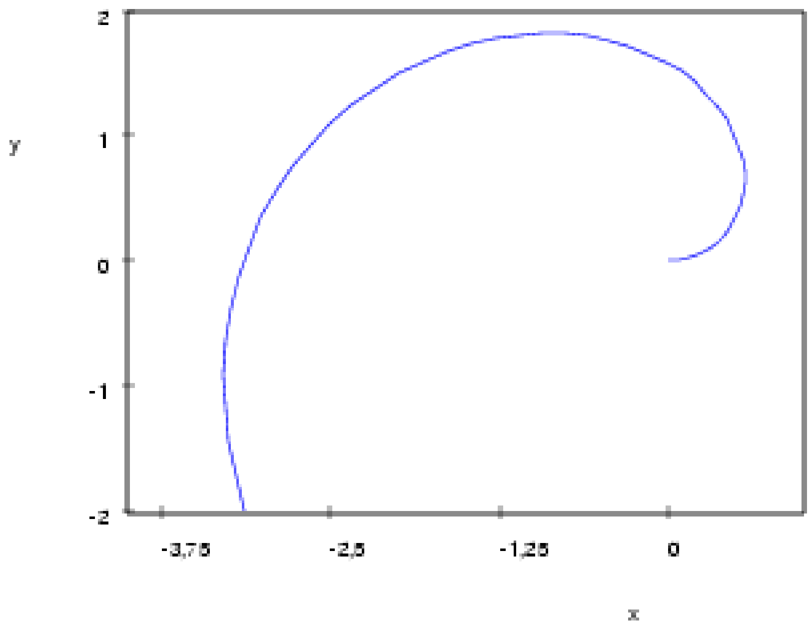
First, this object is plotted without clipping:

```
>> plot2d(Axes = Box, Labeling = TRUE, spiral)
```



In the next plot, the object is clipped to the horizontal range $x \in [-4, 1]$ and the vertical range $y \in [-2, 2]$:

```
>> plot2d(Axes = Box, Labeling = TRUE,
          ViewingBox = [-4..1, -2..2], spiral)
```




```
>> delete spiral:
```

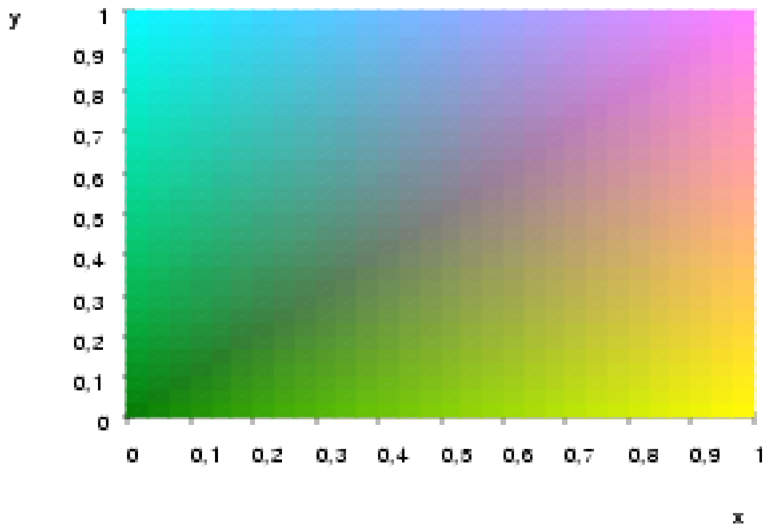
Example 5. We demonstrate user-defined color functions. The following function produces admissible RGB-values between 0 and 1 for objects with coordinates from the range $x \in [0, 1]$ and $y \in [0, 1]$:

```
>> myColor := (x, y) -> [x, 0.5 + abs(x - y)/(1 + x + y), y]:
```

The unit square is to be colored by the function above. We cover the square by $2\,n^2$ triangles, each of which is displayed with a flat color determined by `myColor`:

```
>> n :=  30:
   plot2d([Mode = List,
           [polygon(point((i-1)/n, (j-1)/n),
                    point((i-1)/n,   j  /n),
                    point(  i  /n,   j  /n),
                    Filled = TRUE
                    ) $ i = 1..n $ j = 1..n,
            polygon(point((i-1)/n, (j-1)/n),
                    point(  i  /n, (j-1)/n),
                    point(  i  /n,   j  /n),
```

```
            Filled = TRUE
            ) $ i = 1..n $ j = 1..n
   ],
   Color = [Function, myColor]
  ]):
```




```
>> delete myColor, n:
```

`plot3d` – 3D plots

`plot3d(object1, object2, ...)` generates a 3D plot of graphical objects such as curves, surfaces, points, and polygons.

Call(s):

- `plot3d(<SceneOptions,> object1, object2, ...)`

Parameters:

`object1, object2, ...` — graphical objects as described below

Options:

`SceneOptions` — a sequence of scene options. These determine the general appearance of the graphical scene. See `?plotOptions3d` for details.

Return Value: MuPAD's graphics tool is called to render the graphical scene, and the `null()` object is returned to the MuPAD session.

Related Functions: `plot`, `plotfunc2d`, `plot2d`, `plotfunc3d`

Details:

- `plot3d` is a low level interface to create 3D plots from graphical primitives. For graphs of functions, the specialized routines `plotfunc3d` and `plot::Function3d` are more convenient. For graphical scenes built from primitives, we recommend to use the `plot` library, which provides various primitives and tools. In most cases, the user will find it more convenient to use the `plot` library rather than `plot3d`.

- There are three types of graphical objects that can be plotted by `plot3d`: i) lists of graphical primitives (points and polygons), ii) parametrized curves, and iii) parametrized surfaces.

- i) *Lists of graphical primitives* are objects of the following form:

  `[Mode = List, [primitive1, primitive2, ...] <, Options>]`

  The available primitives are points, polygons and filled polygons generated by the MuPAD functions `point` and `polygon`, respectively. You can use such primitives to build more complicated graphical objects.

  Options are specified by equations `OptionName = value`. The following table gives an overview of the available options:

OptionName	admissible values	default value
Color	[*Flat*], [*Flat*, [r,g,b]], [*Height*], [*Height*, [r,g,b], [R,G,B]], [*Function*, f]	[*Height*]
LineStyle	*SolidLines*, *DashedLines*	*SolidLines*
LineWidth	positive integers	1
PointStyle	*Circles*, *FilledCircles*, *FilledSquares*, *Squares*	*FilledSquares*
PointWidth	positive integers	30
Title	strings	""
TitlePosition	[x, y]	

  See the description below for further details on each option.

- ii) *Parametric curves* are given by a parametrization $u \mapsto [x(u), y(u), z(u)]$ with expressions $x(u)$, $y(u)$, $z(u)$ defining the coordinates as functions of a curve parameter u. In `plot3d`, a curve is defined by an object of the following form:

  `[Mode=Curve, [x(u),y(u),z(u)], u = [umin,umax] <, Options>]`

The parametrization `x(u)`, `y(u)`, `z(u)` consists of arithmetical expressions in one indeterminate `u` (an identifier). They must not contain any other symbolic parameters that cannot be converted to real floating point numbers. The range of the curve parameter `u` is given by the real numbers or numerical expressions `umin` and `umax`.

If the parametrization is given by user-defined functions that accept only numerical values, then premature evaluation can be avoided using `hold(x)(u)`, `hold(y)(u)`, `hold(z)(u)` with the symbolic curve parameter `u`.

Options are specified by equations `OptionName = value`. All options for a list of primitives can be used. For curves, the following additional options are available:

`OptionName`	admissible values	default value
Grid	[integer]	[100]
Smoothness	[integer]	[0]
Style	[*Points*], [*Lines*], [*LinesPoints*], [*Impulses*]	[*Lines*]

See the description below for further details on each option.

iii) *Parametric surfaces* are given by a map $(u, v) \mapsto [x(u, v), y(u, v), z(u, v)]$ with expressions $x(u, v)$, $y(u, v)$, $z(u, v)$ defining the coordinates as functions of two surface parameters u, v. In `plot3d`, a surface is defined by an object of the following form:

```
[Mode = Surface, [x(u, v), y(u, v), z(u, v)],
 u = [umin, umax], v = [vmin, vmax] <, Options>]
```

The parametrization `x(u,v)`, `y(u,v)`, `z(u,v)` consists of arithmetical expressions in two indeterminates `u`, `v` (identifiers). They must not contain any other symbolic parameters that cannot be converted to real floating point numbers. The ranges of the surface parameters `u` and `v` are given by the real numbers or numerical expressions `umin`, `umax` and `vmin`, `vmax`, respectively.

If the parametrization is given by user-defined functions that accept only numerical values, then premature evaluation can be avoided using `hold(x)(u,v)`, `hold(y)(u,v)`, `hold(z)(u,v)` with the symbolic surface parameters `u`, `v`.

Options are specified by equations `OptionName = value`. All options for a list of primitives can be used. For surfaces, the following additional options are available:

`OptionName`	admissible values	default value
Grid	[integer, integer]	[20, 20]
Smoothness	[integer, integer]	[0, 0]

OptionName	admissible values	default value
Style	[*Points*] [*WireFrame* , *Mesh*] [*WireFrame* , *ULine*] [*WireFrame* , *VLine*] [*HiddenLine* , *Mesh*] [*HiddenLine* , *ULine*] [*HiddenLine* , *VLine*] [*ColorPatches* , *Only*] [*ColorPatches* , *AndMesh*] [*ColorPatches* , *AndULine*] [*ColorPatches* , *AndVLine*] [*Transparent* , *Only*] [*Transparent* , *AndMesh*] [*Transparent* , *AndULine*] [*Transparent* , *AndVLine*]	[*ColorPatches* , *AndMesh*]

See the description below for further details on each option.

- *The graph of a function* $f(x, y)$ can be plotted as a parametrized surface

```
[Mode = Surface, [x, y, f(x, y)], x = [xmin, xmax],
 y = [ymin, ymax] <, Options>]:
```

However, it is more convenient to use `plotfunc3d` or `plot::Function3d` to plot or generate function graphs.

- MuPAD graphics can be saved in a variety of graphical formats. In a `plot3d` command, the *PlotDevice* scene option allows to specify the conversion into the two MuPAD specific formats '*Ascii*' and '*Binary*'. See the help page `plotOptions3d` for details.

  For graphical standard formats such as *Postscript*, *JPEG*, *TIFF* etc., no direct conversion is available by a plot command inside a MuPAD session. Instead, conversion has to be requested interactively via the graphical interface of the rendering tool VCam. In a MuPAD Pro notebook, double click on the graphics to activate this interface. Using the menu item "Edit/Save Graphics ..", you can choose the desired format in the "Export Graphics" dialog box.

Option <`Color = value`>**:**

- This option determines the color of the object. Admissible values are [*Flat*], [*Flat* , [r,g,b]], [*Height*], [*Height* , [r,g,b] , [R,G,B]] and [*Function* , f]. The default is *Color* = [*Height*].

  - With *Color* = [*Flat*], the object is displayed with a flat color. The actual color is chosen automatically.

- With *Color* = [*Flat* , [r, g, b]], the object is displayed with a flat color. The values `r`, `g`, `b` represent the red, green and blue contributions according to the RGB color model. They must be real numbers between `0` and `1`. Pre-defined colors are provided by MuPAD's `RGB` data structure.

- With *Color* = [*Height*], the color varies with the y-coordinate. The actual colors are chosen automatically.

- With *Color* = [*Height* , [r, g, b], [R, G, B]], the color varies with the y-coordinate. The parts of the object with small values of y are displayed with the color `[r, g, b]`, parts with large values of y are displayed with the color `[R, G, B]`. Interpolated color values are used in between.

- With *Color* = [*Function* , f], users may implement their own coloring scheme. The parameter `f` must be a MuPAD procedure returning a color as a list `[r, g, b]`.

  - Inside a list of primitives, the function `f` must accept three parameters:

    `f := proc(x, y, z) begin ..; return([r, g, b]) end:`

    During the numerical evaluation of the plot this function is called with arguments `(x, y, z)` from the viewing range of the object.

    Note that polygons are always displayed with a flat color.

  - Inside a curve object, the function `f` must accept four parameters:

    `f := proc(x, y, z, u) begin ..; return([r, g, b]) end:`

    During the numerical evaluation of the plot this function is called with the arguments `(x(u), y(u), z(u), u)`, where `u` is the curve parameter and `x(u)`, `y(u)`, `z(u)` are the corresponding coordinates.

  - Inside a surface object, the function `f` must accept five parameters:

    `f := proc(x, y, z, u, v) begin ..; return([r, g, b]) end:`

    During the numerical evaluation of the plot this function is called with the arguments `(x(u,v), y(u,v), z(u,v), u, v)`, where `u`, `v` are the curve parameters and `x(u,v)`, `y(u,v)`, `z(u,v)` are the corresponding coordinates.

If the color function `f` is created inside a procedure, using local variables of this procedure, then this procedure must use `option escape`.

Option <`Grid = [n] (for curves)` >:

- Inside curve objects, this option determines the number of sample points. The graphics uses linear interpolation between adjacent sample points. The integer `n` must be larger than 1; the default is *`Grid`* = `[100]`. Large values of `n` generate a smooth curve. Alternatively, the *`Smoothness`* parameter can be increased.

Option <`Grid = [nu, nv] (for surfaces)` >:

- Inside surface objects, this option determines the number of sample points for the surface parameters `u` and `v`. The graphics uses linear interpolation between adjacent sample points. The integers `nu`, `nv` must be larger than 1; the default is *`Grid`* = `[20, 20]`. Large values of `nu`, `nv` generate a smooth surface. Alternatively, the *`Smoothness`* parameters can be increased.

Option <`LineStyle = value`>:

- This option determines the style in which lines are displayed. Admissible values are *`SolidLines`* and *`DashedLines`*; the default is *`LineStyle`* = *`SolidLines`*.

Option <`LineWidth = n`>:

- This option sets the width of the lines belonging to the object. Admissible values for `n` are nonnegative integers; the default is *`LineWidth`* = 1.

Option <`PointStyle = value`>:

- This option sets the style in which point objects are displayed. Admissible values are *`Circles`*, *`Squares`*, *`FilledCircles`*, and *`FilledSquares`*. The default is *`PointStyle`* = *`FilledSquares`*.

Option <`PointWidth = n`>**:**

This option sets the size of point objects. Admissible values for `n` are positive integers; the default is *`PointWidth`* = 30.

Option <`Smoothness = [n] (for curves)` >**:**

Inside curve objects, this option determines the number of additional interpolation points between the sample points of the curve parameter determined by the *`Grid`* option. Admissible values for `n` are integers between 0 and 20; the default is *`Smoothness`* = `[0]`. Lines are depicted as linear segments connecting these interpolation points. Consequently, large values of `n` produce smooth lines.

Option <`Smoothness = [nu, nv] (for surfaces)` >**:**

Inside surface objects, this option determines the number of interpolation points between the sample points of the surface parameters determined by the *`Grid`* option. Linear interpolation is used between interpolation points. Admissible values for `nu`, `nv` are integers between 0 and 20; the default is *`Smoothness`* = `[0, 0]`. Large values of `nu`, `nv` generate a smooth surface.

Option <`Style = value (for curves)` >**:**

This option sets the style in which curves are displayed. Admissible values are `[`*`Points`*`]`, `[`*`Lines`*`]`, `[`*`LinesPoints`*`]` and `[`*`Impulses`*`]`. The default is *`Style`* = `[`*`Lines`*`]`.

- With *`Style`* = `[`*`Points`*`]`, only the sample points determined by the *`Grid`* option are displayed.
- With *`Style`* = `[`*`Lines`*`]`, the curve is displayed as a collection of line segments connecting the sample points.
- With *`Style`* = `[`*`LinesPoints`*`]`, both the sample points as well as the connecting line segments are displayed.
- With *`Style`* = `[`*`Impulses`*`]`, the curve is displayed like a "histogram": vertical lines from the bottom of the scene to the sample points are drawn.

Option <`Style = value (for surfaces)` >:

This option sets the style in which parametrized surfaces are displayed. The default is `Style` = `[ColorPatches, AndMesh]`.

- With `Style` = `[Points]`, only the sample points determined by the `Grid` option are displayed.
- With `Style` = `[WireFrame, Mesh]`, a wireframe with the parameter lines of both surface parameters is displayed.
- With `Style` = `[WireFrame, ULine]`, a wireframe consisting of the parameter lines of the parameter `u` is displayed.
- With `Style` = `[WireFrame, VLine]`, a wireframe consisting of the parameter lines of the parameter `v` is displayed.
- With `Style` = `[HiddenLine, Mesh]`, the surface is displayed as an opaque object. Additionally, the parameter lines of both parameters are displayed.
- With `Style` = `[HiddenLine, ULine]`, the surface is displayed as an opaque object. Additionally, the parameter lines of the parameter `u` are displayed.
- With `Style` = `[HiddenLine, VLine]`, the surface is displayed as an opaque object. Additionally, the parameter lines of the parameter `v` are displayed.
- With `Style` = `[ColorPatches, Only]`, the surface is displayed as an opaque object. All surface patches are colored. No parameter lines are displayed.
- With `Style` = `[ColorPatches, AndMesh]`, the surface is displayed as an opaque object. All surface patches are colored. Additionally, the parameter lines of both parameters are displayed.
- With `Style` = `[ColorPatches, AndULine]`, the surface is displayed as an opaque object. All surface patches are colored. Additionally, the parameter lines of the parameter `u` are displayed.
- With `Style` = `[ColorPatches, AndVLine]`, the surface is displayed as an opaque object. All surface patches are colored. Additionally, the parameter lines of the parameter `v` are displayed.
- With `Style` = `[Transparent, Only]`, the surface patches are filled with patterns, simulating semi-transparency. No parameter lines are displayed.

- With *Style* = [*Transparent* , *AndMesh*], the surface patches are filled with patterns, simulating semi-transparency. Additionally, the parameter lines of both parameters are displayed.
- With *Style* = [*Transparent* , *AndULine*], the surface patches are filled with patterns, simulating semi-transparency. Additionally, the parameter lines of the parameter `u` are displayed.
- With *Style* = [*Transparent* , *AndVLine*], the surface patches are filled with patterns, simulating semi-transparency. Additionally, the parameter lines of the parameter `v` are displayed.

Please note that the *Style* option *Transparent* is *not* available under Windows. When *Transparent* is choosen it is internally changed to *ColorPatches* automatically.

Option <`Title = TitleString` >**:**

This option adds the text given by the string `TitleString` to the object. The default is the empty string *Title* = `""`, i.e., no title.

Option <`TitlePosition = [x, y]`>**:**

This option determines the position of the object title. The parameters `x`, `y` must be numerical values between 0 and 10. The position `[0, 0]` denotes the upper left corner of the scene, the position `[10, 10]` denotes the lower right corner.

Note that the specified positions are relative to the entire scene. Consequently, if titles are specified for several objects, their positions should differ to avoid overlap.

Object titles can be moved interactively with the mouse to any appropriate position inside the scene.

Example 1. We demonstrate plotting of graphical primtives. First, three point primitives, a line primitive and a filled polygon are defined:

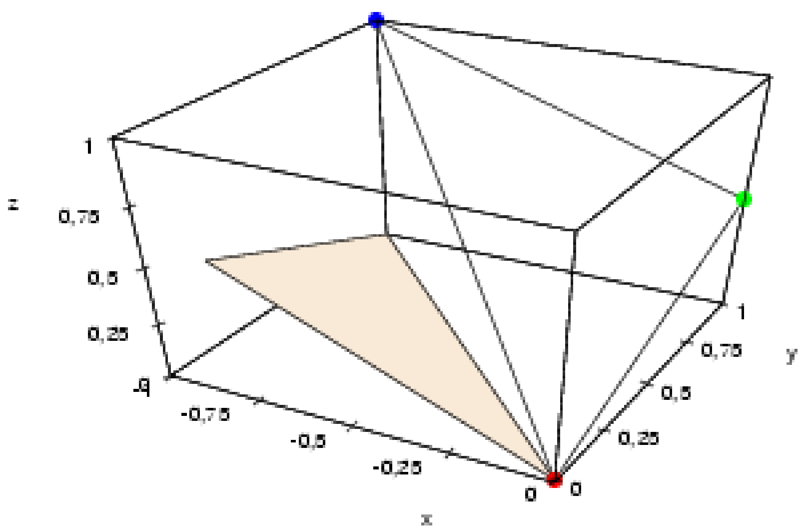
```
>> p1 := point(0, 0, 0, Color = RGB::Red):
   p2 := point(0, 1, 1/2, Color = RGB::Green):
   p3 := point(-1, 1, 1, Color = RGB::Blue):
   line := polygon(point(0, 0, 0), point(0, 1, 1/2),
                   point(-1, 1, 1), Closed = TRUE,
                   Color = RGB::Black):
   triangle := polygon(point(0, 0, 0), point(-1, 0.2, 0.4),
                       point(-1, 1, 0), Closed = TRUE,
                       Filled = TRUE, Color = RGB::Antique):
```

These are combined to a graphical object:

```
>> object := [Mode = List, [p1, p2, p3, line, triangle]]:
```

Finally, this object is plotted:

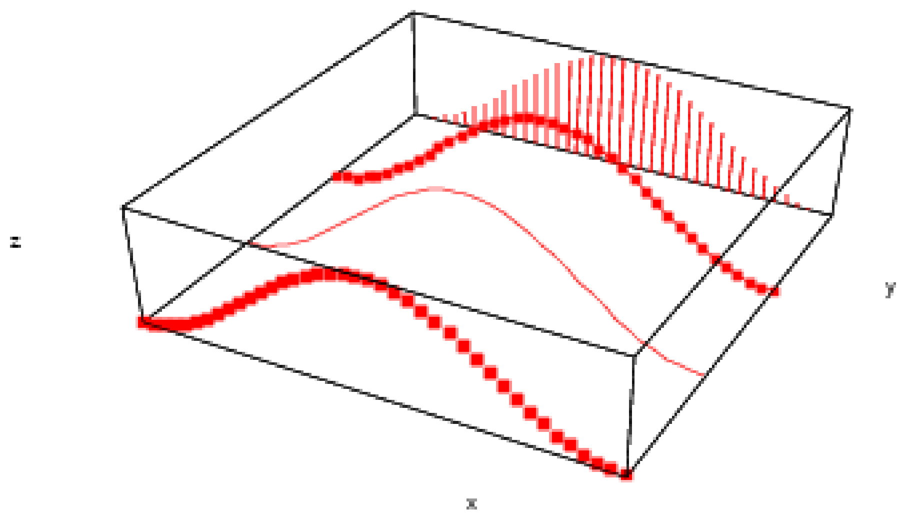
```
>> plot3d(BackGround = RGB::White, ForeGround = RGB::Black,
          PointWidth = 70, PointStyle = FilledCircles,
          Axes = Box, object)
```




```
>> delete p1, p2, p3, line, triangle, object:
```

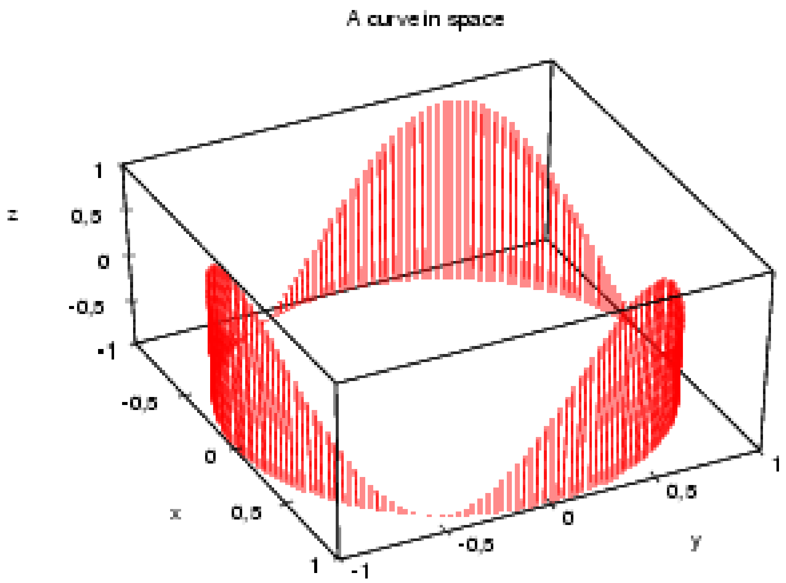
Example 2. We plot curves. The following picture demonstrates various styles:

```
>> plot3d(Axes = Box, Ticks = 0,
          BackGround = RGB::White, ForeGround = RGB::Black,
          [Mode = Curve, [u, -PI, cos(u)], u = [-PI, PI],
           Grid = [40], Style = [Points], PointWidth = 40
          ],
          [Mode = Curve, [u, -PI/3, cos(u)], u = [-PI, PI],
           Grid = [40], Style = [Lines]
          ],
          [Mode = Curve, [u, PI/3, cos(u)], u = [-PI, PI],
           Grid = [40], Style = [LinesPoints], PointWidth = 30
          ],
          [Mode = Curve, [u, PI, cos(u)], u = [-PI, PI],
           Grid = [40], Style = [Impulses]
          ]):
```



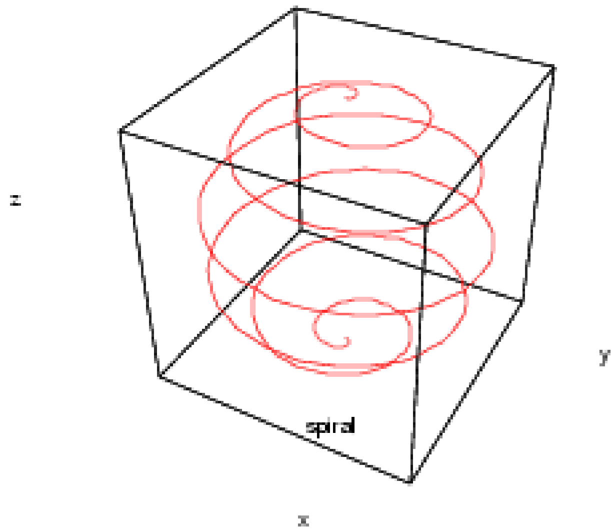
The following command plots a “histogram style” graph of the cosine function defined over the unit circle in the x-y-plane:

```
>> plot3d(Axes = Box, Ticks = 5, CameraPoint = [20, -10, 30],
          BackGround = RGB::White, ForeGround = RGB::Black,
          Labeling = TRUE, Labels = ["x", "y", "z"],
          Title = "A curve in space",
          [Mode = Curve, [cos(u), sin(u), sin(3*u)], u = [0, 2*PI],
           Grid = [200], Style = [Impulses]
          ])
```



The following command plots a spiral on the unit sphere:

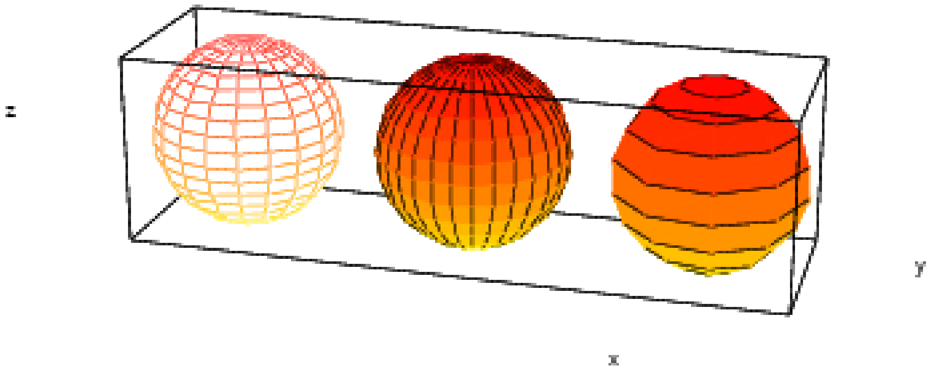
```
>> plot3d(Axes = Box, Ticks = 0, Scaling = Constrained,
          Title = "spiral", TitlePosition = Below,
          [Mode = Curve,
           [cos(12*u*PI)*sin(u*PI),
            sin(12*u*PI)*sin(u*PI),
            cos(u*PI)],
           u = [0, 1], Grid = [50], Smoothness = [5]
          ])
```



Example 3. We demonstrate surface plots. The next command generates spheres of radius 1 parametrized by polar coordinates. It illustrates various surface styles:

```
>> plot3d(Axes = Box, Ticks = 0, Scaling = Constrained,
          BackGround = RGB::White, ForeGround = RGB::Black,
          CameraPoint = [6, -21, 8],
          [Mode = Surface,
           [-2.5 + sin(u)*cos(v), sin(u)*sin(v), cos(u)],
           u = [0, PI], v = [0, 2*PI],
           Grid = [20, 20], Smoothness = [0, 0],
           Style = [HiddenLine, Mesh]
          ],
          [Mode = Surface,
           [sin(u)*cos(v), sin(u)*sin(v), cos(u)],
           u = [0, PI], v = [0, 2*PI],
           Grid = [15, 30], Smoothness = [0, 0],
           Style = [ColorPatches, AndULine]
          ],
          [Mode = Surface,
           [2.5 + sin(u)*cos(v), sin(u)*sin(v), cos(u)],
           u = [0, PI], v = [0, 2*PI],
           Grid = [10, 10], Smoothness = [0, 0],
```

```
 Style = [Transparent, AndVLine]
])
```

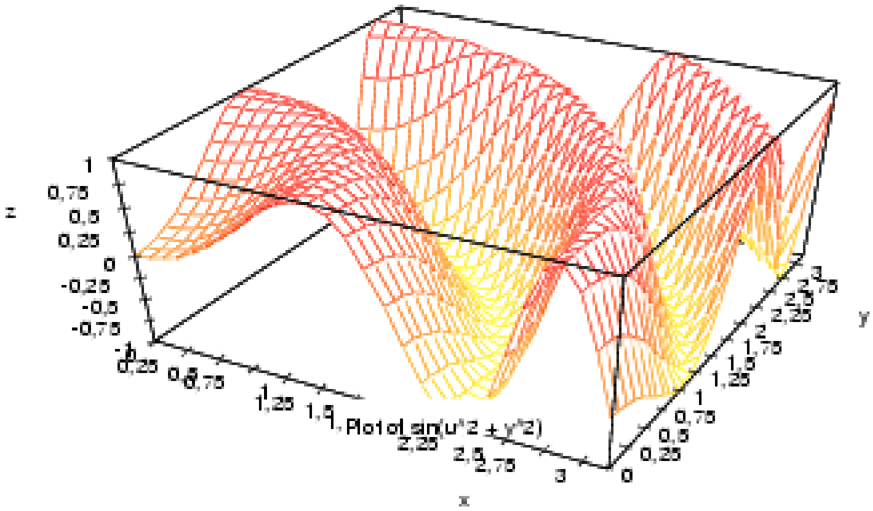


The effect of the options *Grid* and *Smoothness* is demonstrated by discs in the x-y-plane:

```
>> plot3d(Axes = None, Scaling = Constrained,
          BackGround = RGB::White, ForeGround = RGB::Black,
          CameraPoint = [0, -1, 20],
          [Mode = Surface, [-2.5 + v*sin(u), v*cos(u), 0],
           u = [-PI, PI], v = [0, 1], Style = [WireFrame, Mesh],
           Grid = [ 6,  6], Smoothness = [0, 0]
          ],
          [Mode = Surface, [v*sin(u), v*cos(u), 0],
           u = [-PI, PI], v = [0, 1], Style = [WireFrame, Mesh],
           Grid = [ 6,  6], Smoothness = [3, 2]
          ],
          [Mode = Surface, [2.5 + v*sin(u), v*cos(u), 0],
           u = [-PI, PI], v = [0, 1], Style = [WireFrame, Mesh],
           Grid = [20, 10], Smoothness = [0, 0]
          ])
```

The graph of a function is plotted as a parametrized surface:

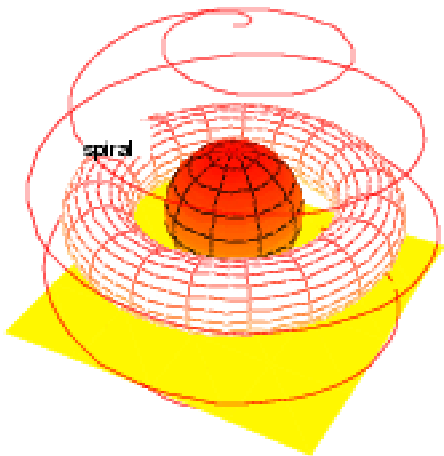
```
>> plot3d(Axes = Box, Ticks = 8,
          BackGround = RGB::White, ForeGround = RGB::Black,
          Title = "Plot of sin(u^2 + v^2)", TitlePosition = Below,
          [Mode = Surface, [u, v, sin(u^2 + v^2)],
           u = [0, PI], v = [0, PI],
           Grid = [30, 30], Style = [HiddenLine, Mesh]
          ])
```



Various objects of different type are combined to a graphical scene:

```
>> plot3d(Axes = None, Scaling = Constrained,
          BackGround = RGB::White, ForeGround = RGB::Black,
          Title = "Three surfaces and a curve",
          TitlePosition = Below,
          CameraPoint = [13, -24, 20],
          [Mode = Surface,
           [(4 + cos(v))*cos(u), (4 + cos(v))*sin(u), sin(v)],
           u = [0, 2*PI], v = [0, 2*PI],
           Grid = [20, 20], Smoothness = [2, 0],
           Style = [HiddenLine, Mesh]
          ],
          [Mode = Surface,
           [2*cos(u)*sin(v), 2*sin(u)*sin(v), 2*cos(v)],
           u = [0, 2*PI], v = [0, PI],
           Grid = [10, 10], Smoothness = [2, 2],
           Style = [ColorPatches, AndMesh]
          ],
          [Mode = Surface, [u, v, -3], u = [-5, 5], v = [-5, 5],
           Grid = [5, 5], Smoothness = [0, 0],
           Style = [ColorPatches, Only]
          ],
          [Mode = Curve,
           [6*cos(12*u)*sin(u), 6*sin(12*u)*sin(u), 6*cos(u)],
```

```
 u = [0, PI], Grid = [50], Smoothness = [5],
 Title = "spiral"
])
```



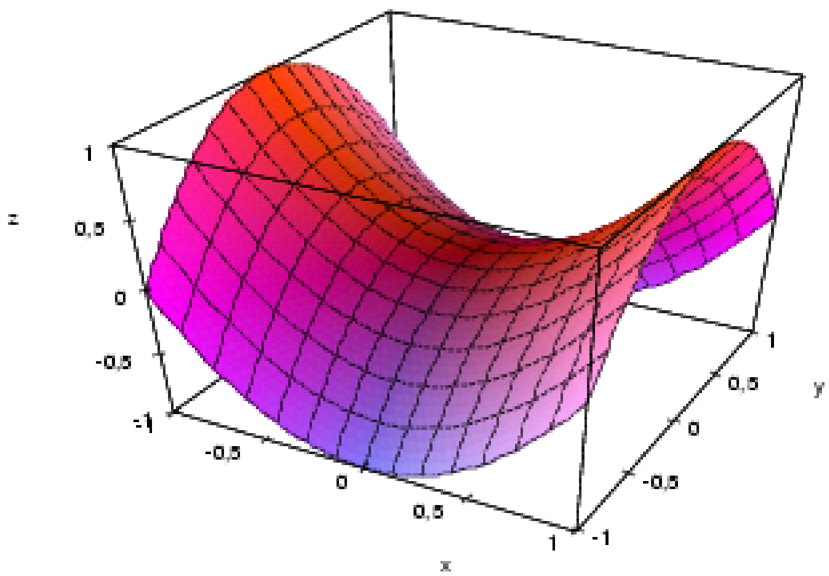
Three surfaces and a curve

Example 4. We demonstrate user-defined color functions. The following function produces admissible RGB-values between 0 and 1 for objects with coordinates $x, y, z \in [-1, 1]$:

```
>> myColor := (x, y, z, u, v) ->
              [(abs(x) + 1)/2, abs(x - y)/(3 + z), abs(y)]:
```

A hyperboloid over the unit square is to be colored by the function above. We plot the graph of the function $(x, y) \mapsto x^2 - y^2$ as a parametrized surface:

```
>> plot3d(Axes = Box,
          BackGround = RGB::White, ForeGround = RGB::Black,
          [Mode = Surface, [x, y, x^2 - y^2],
           x = [-1, 1], y = [-1, 1],
           Grid = [15, 15], Smoothness = [3, 3],
           Style = [ColorPatches, AndMesh],
           Color = [Function, myColor]
          ])
```



>> delete myColor:

---

plotfunc2d – **2D plots of function graphs**

plotfunc2d(f1, f2, ...) generates a 2D plot of the graphs of the univariate functions f1, f2 etc.

Call(s):

- plotfunc2d(<SceneOptions,> f1, f2, ... <, *Grid* = n>)
- plotfunc2d(<SceneOptions,> f1, f2, ..., x = xmin..xmax <, *Grid* = n>)
- plotfunc2d(<SceneOptions,> f1, f2, ..., x = xmin..xmax, y = ymin..ymax <, *Grid* = n>)

Parameters:

`f1, f1, ...`	—	the functions: arithmetical expressions or `piecewise` objects containing one indeterminate `x`
`x`	—	the horizontal coordinate: an identifier
`xmin, xmax`	—	the horizontal plot range: finite real numerical expressions
`y`	—	a dummy name for the vertical coordinate: an identifier. This name is used to label the y-axis.
`ymin, ymax`	—	the vertical plot range: finite real numerical expressions

Options:

`SceneOptions`	—	a sequence of scene options. These determine the general appearance of the graphical scene. See `?plotOptions2d` for details.
Grid `= n`	—	sets the number of sample points used for the plot. The integer `n` must be larger than 1; the default is *Grid* = 100.

Return Value: MuPAD's graphics tool is called to render the graphical scene. The `null()` object is returned to the MuPAD session.

Related Functions: `plot`, `plot::Function2d`, `plot2d`, `plot3d`, `plotfunc3d`

Details:

- The functions must not contain any symbolic parameters apart from `x` that cannot be converted to floating point values.
- If no horizontal plot range is specified, the default range `x = -5..5` is used.
- If a vertical range `y = ymin..ymax` is specified, only function values between `ymin` and `ymax` are displayed. The name `y` of the vertical coordinate is arbitrary: any identifier may be used.
- Non-real function values are ignored. Cf. example 2.
- Functions with singularities are handled. Cf. example 3.
- Discontinuities and piecewise defined functions are handled. Cf. examples 5, 6.
- The graph of a function $f(x)$ can also be plotted by `plot2d` as a parametrized curve

  `[`*Mode* `=` *Curve* `, [x, f(x)], x = [xmin, xmax] <, Options>]:`

  This way, ranges, color options and style options can be specified separately for each function. See the help page of `plot2d` for details.

- The `plot` library provides the routine `plot::Function2d` which allows to create a function graph as a graphical primitive, and to combine it with other graphical objects.

- MuPAD graphics can be saved in a variety of graphical formats. In a `plotfunc2d` command, the *PlotDevice* scene option allows to specify the conversion into the two MuPAD specific formats '*Ascii*' and '*Binary*'. See the help page `plotOptions2d` for details.

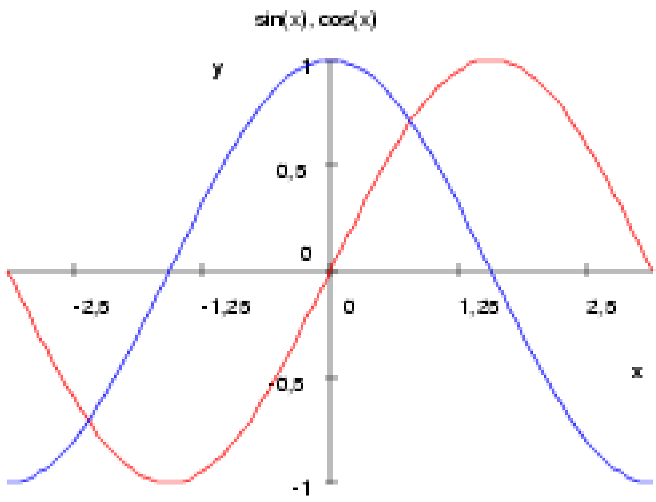
  For graphical standard formats such as *Postscript*, *JPEG*, *TIFF* etc., no direct conversion is available by a plot command inside a MuPAD session. Instead, conversion has to be requested interactively via the graphical interface of the rendering tool VCam. In a MuPAD Pro notebook, double click on the graphics to activate this interface. Using the menu item "Edit/Save Graphics ..", you can choose the desired format in the "Export Graphics" dialog box.

Option <`Grid = n`>:

- This option determines the number of sample points (function evaluations). The graphics uses linear interpolation between adjacent sample points. The integer `n` must be larger than 1; the default is *Grid* = `100`. Large values of `n` generate a smooth graph.

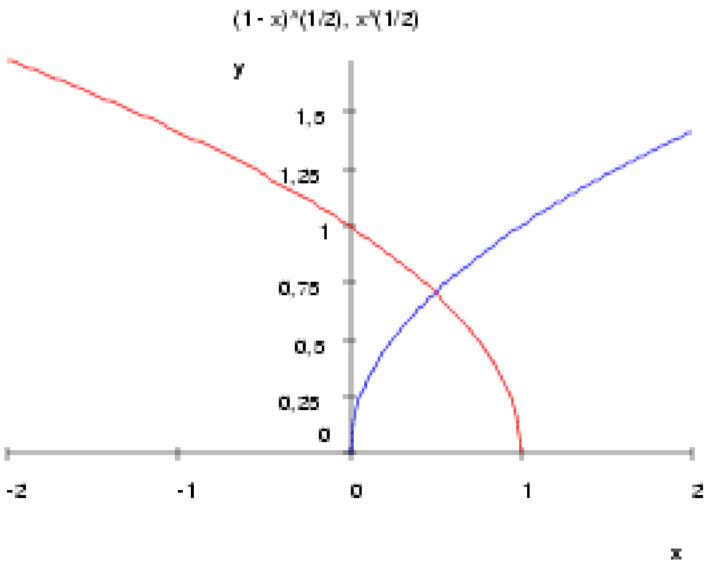
Example 1. The following command draws the sine and the cosine functions on the interval $[-\pi, \pi]$:

```
>> plotfunc2d(sin(x), cos(x), x = -PI..PI):
```



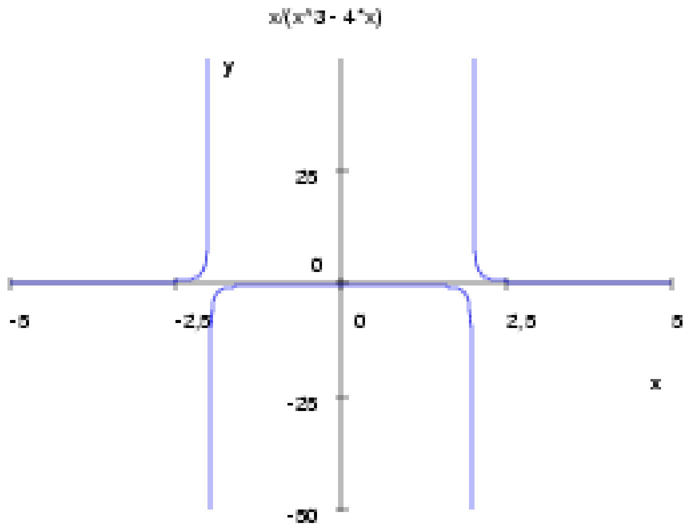
Example 2. Only real functions values are plotted:

```
>> plotfunc2d(sqrt(1 - x), sqrt(x), x = -2..2):
```

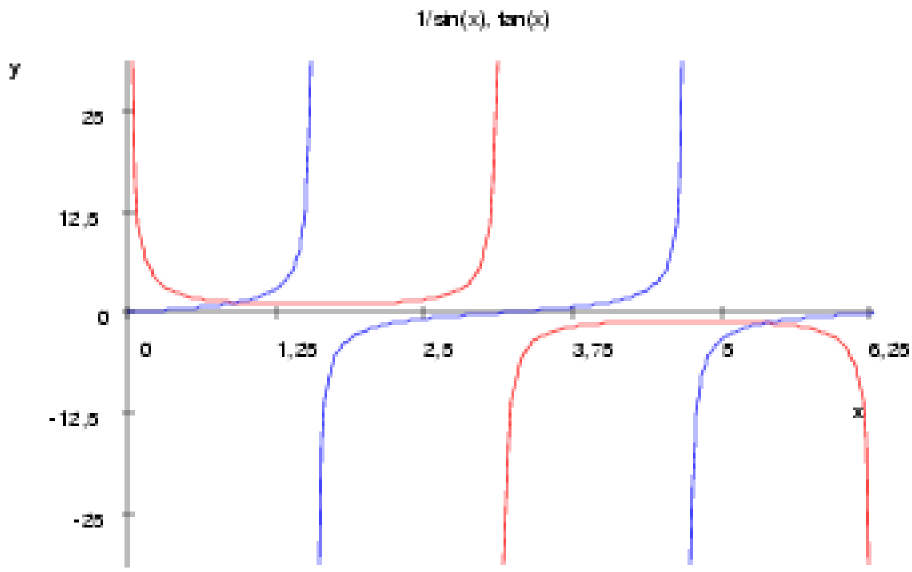


Example 3. The following functions have singularities in the specified interval:

```
>> plotfunc2d(x/(x^3 - 4*x), x = -5..5):
```

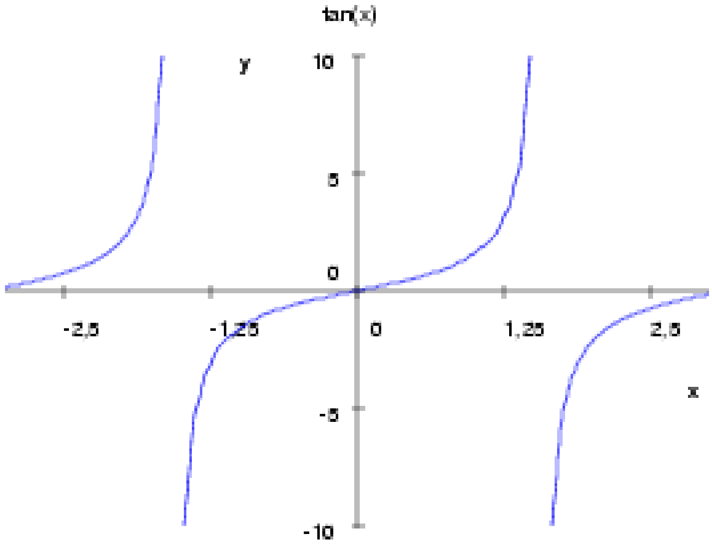



```
>> plotfunc2d(1/sin(x), tan(x), x = 0..2*PI):
```



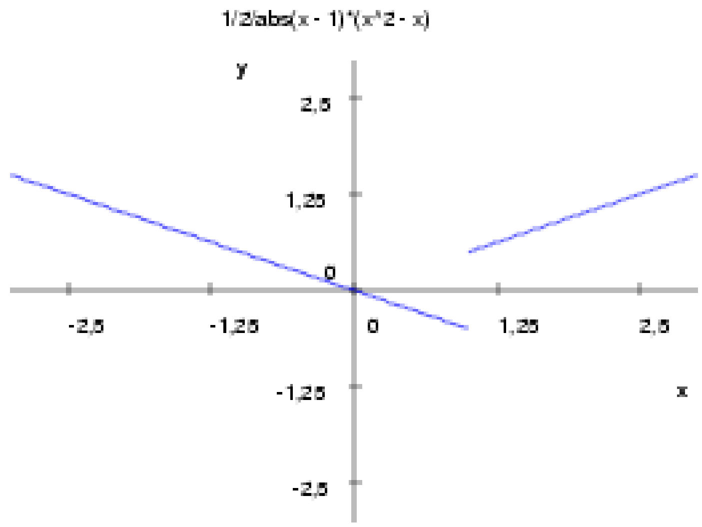
Example 4. We define a vertical range to which the function graph is restricted:

```
>> plotfunc2d(tan(x), x = -3..3, y = -10..10):
```



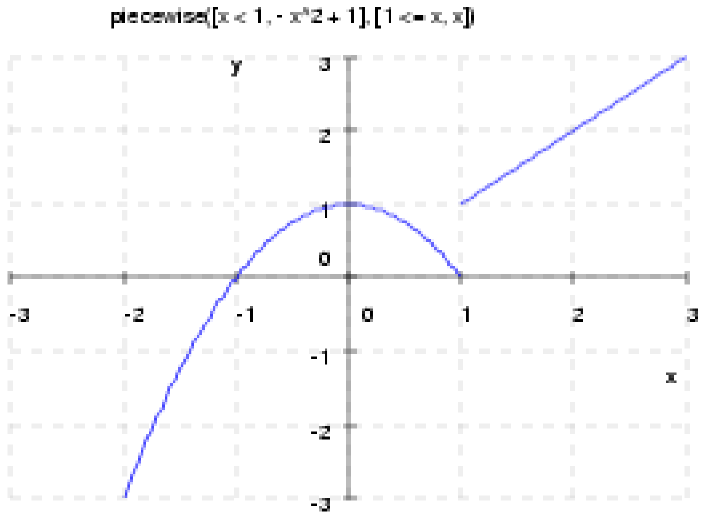
Example 5. The following function has a jump discontinuity:

```
>> plotfunc2d((x^2 - x)/(2*abs(x - 1)), x = -3..3, y = -3..3)
```

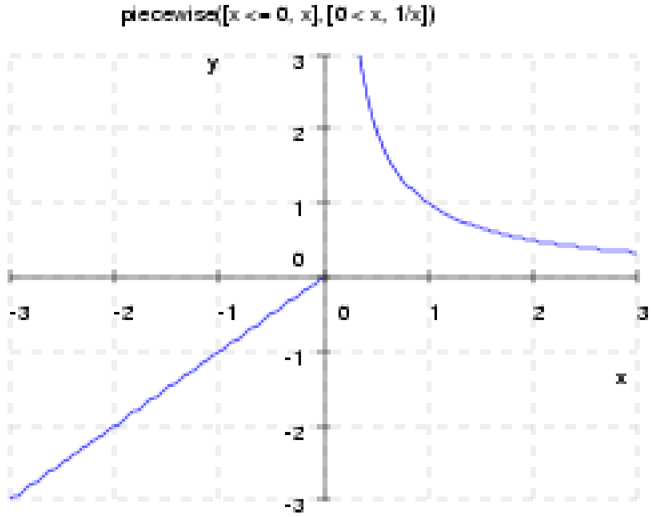


Example 6. Piecewise defined functions are handled:

```
>> f := piecewise([x < 1, -x^2 + 1], [x >= 1, x]):
   plotfunc2d(BackGround = RGB::White,
              ForeGround = RGB::Black,
              GridLines = Automatic,
              Ticks = [Steps = 1, Steps = 1],
              f(x), x = -3..3, y = -3..3)
```



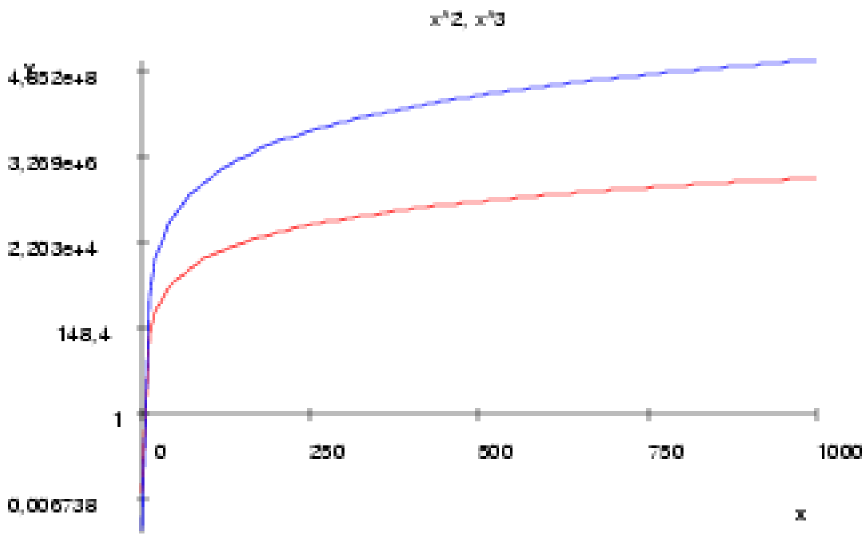

```
>> f := piecewise([x <= 0, x], [x > 0, 1/x]):
   plotfunc2d(BackGround = RGB::White,
              ForeGround = RGB::Black,
              GridLines = Automatic,
              Ticks = [Steps = 1, Steps = 1],
              f(x), x = -3..3, y = -3..3)
```




```
>> delete f:
```

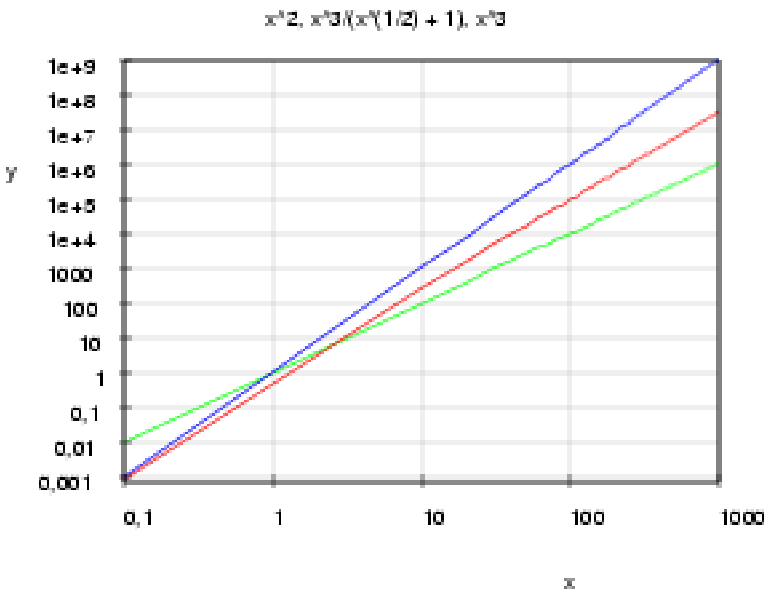
Example 7. We use the scene option *AxesScaling* to create a logarithmic plot:

```
>> plotfunc2d(AxesScaling = [Lin, Log], x^2, x^3, x = 1/10..10^3):
```



We demonstrate various further scene options in a doubly logarithmic plot:

```
>> plotfunc2d(Axes = Box,
              AxesScaling = [Log, Log],
              Discont = FALSE,
              BackGround = RGB::White,
              ForeGround = RGB::Black,
              GridLines = Automatic,
              GridLinesStyle = SolidLines,
              GridLinesColor = RGB::Gray,
              Ticks = [[10^i $ i = -1..3], [10^i $ i = -3..9]],
              x^2, x^3/(1 + x^(1/2)), x^3, x = 1/10..10^3):
```



`plotfunc3d` – 3D plots of function graphs

`plotfunc3d(f1, f2, ...)` generates a 3D plot of the graphs of the bivariate functions `f1`, `f2` etc.

Call(s):

- `plotfunc3d(<SceneOptions,> f1, f2, ... <, Grid = [nx, ny]>)`
- `plotfunc3d(<SceneOptions,> f1, f2, ..., x = xmin..xmax <, Grid = [nx, ny]>)`
- `plotfunc3d(<SceneOptions,> f1, f2, ..., x = xmin..xmax, y = ymin..ymax <, Grid = [nx, ny]>)`

Parameters:

`f1, f1, ...`	— the functions: arithmetical expressions or `piecewise` objects containing two indeterminates `x`, `y`
`x, y`	— the independent variables: identifiers
`xmin, xmax`	— the plot range for `x`: finite real numerical expressions
`ymin, ymax`	— the plot range for `y`: finite real numerical expressions

Options:

`SceneOptions`	—	a sequence of scene options. These determine the general appearance of the graphical scene. See `?plotOptions3d` for details.
Grid = `[nx, ny]`	—	sets the number of sample points in the x and y direction. The integers `nx`, `ny` must be larger than 1; the default is *Grid* = `[20, 20]`.

Return Value: MuPAD's graphics tool is called to render the graphical scene. The `null()` object is returned to the MuPAD session.

Related Functions: `plot`, `plot::Function3d`, `plot2d`, `plot3d`, `plotfunc2d`

Details:

- The functions must not contain any symbolic parameters apart from `x` and `y` that cannot be converted to floating point values.

- If no plot range is specified, the default ranges `x = -5..5` and `y = -5..5` are used.

- Piecewise defined functions are handled. Cf. example 4.

- The graph of a function $f(x, y)$ can also be plotted by `plot3d` as a parametrized surface:

```
[Mode = Surface, [x, y, f(x, y)], x = [xmin, xmax],
 y = [ymin, ymax] <, Options>]:
```

  This way ranges, color options, style options etc. can be specified separately for each function. See the help page of `plot3d` for details.

- The `plot` library provides the routine `plot::Function3d` which allows to create a function graph as a graphical primitive, and to combine it with other graphical objects.

- MuPAD graphics can be saved in a variety of graphical formats. In a `plotfunc3d` command, the *PlotDevice* scene option allows to specify the conversion into the two MuPAD specific formats '*Ascii*' and '*Binary*'. See the help page `plotOptions3d` for details.

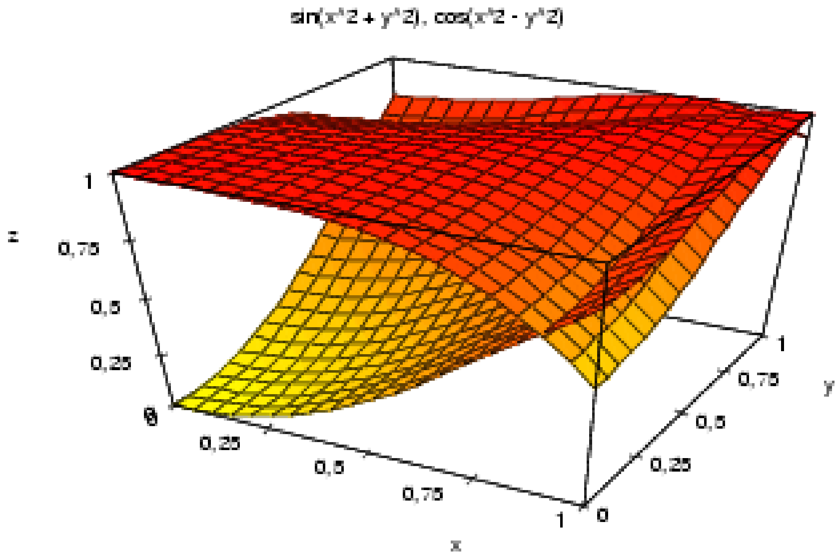
  For graphical standard formats such as *Postscript*, *JPEG*, *TIFF* etc., no direct conversion is available by a plot command inside a MuPAD session. Instead, conversion has to be requested interactively via the graphical interface of the rendering tool VCam. In a MuPAD Pro notebook, double click on the graphics to activate this interface. Using the menu item "Edit/Save Graphics ..", you can choose the desired format in the "Export Graphics" dialog box.

Option <`Grid = [nx, ny]`>**:**

- This option determines the number of sample points in the x and y direction. The graphics uses linear interpolation between adjacent sample points. The integers `nx`, `ny` must be larger than 1; the default is *Grid* = `[20, 20]`. Large values of `nx`, `ny` generate a smooth graph.

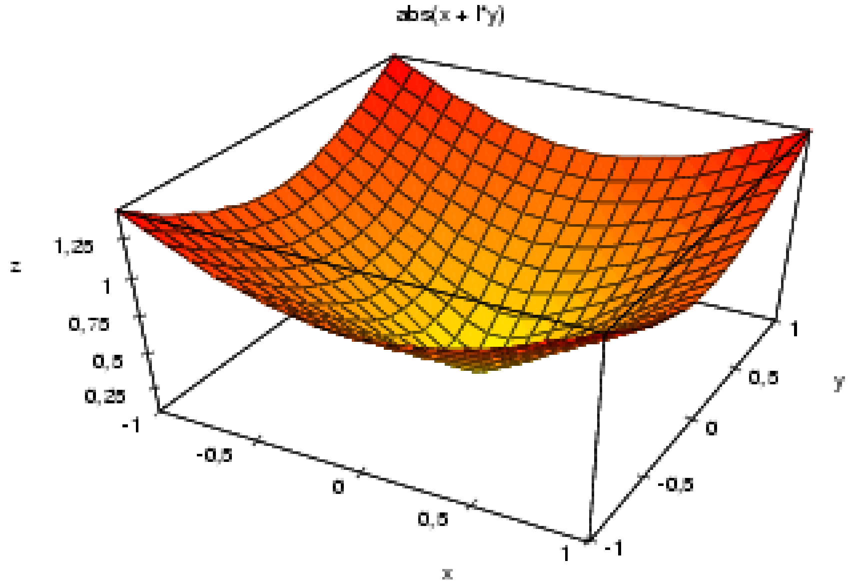
Example 1. The following command draws two functions over the unit square:

```
>> plotfunc3d(BackGround = RGB::White,
              ForeGround = RGB::Black,
              Axes = Box,
              sin(x^2 + y^2), cos(x^2 - y^2),
              x = 0..1, y = 0..1):
```

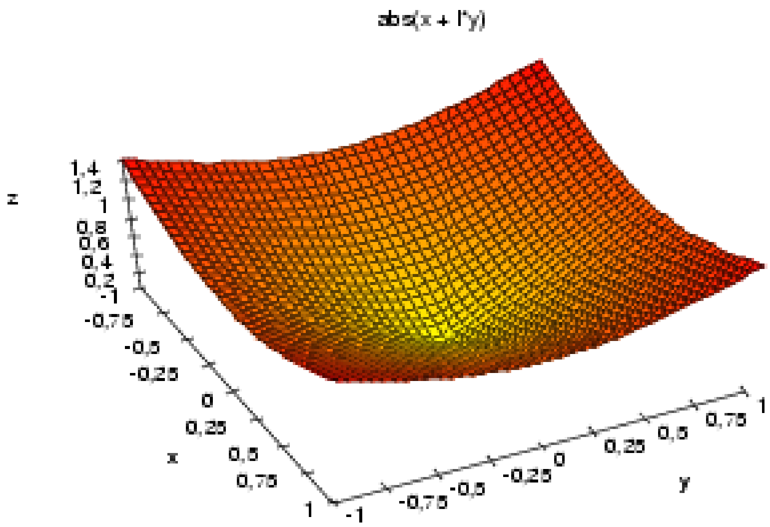


Example 2. We demonstrate the effect of various scene options:

```
>> plotfunc3d(Axes = Box, Ticks = 5,
              abs(x + I*y), x = -1..1, y = -1..1)
```




```
>> plotfunc3d(Arrows = FALSE, Axes = Corner, Ticks = 8,
              Grid = [40, 40], CameraPoint = [10, -5, 15],
              abs(x + I*y), x = -1..1, y = -1..1)
```



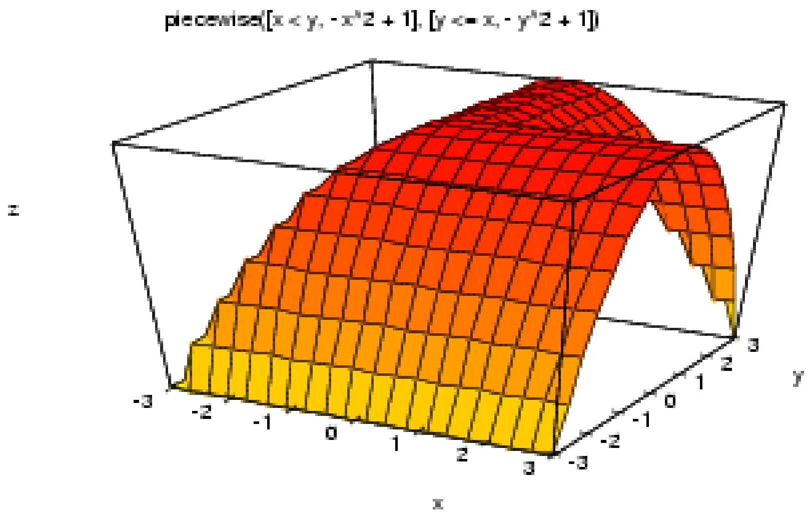
Example 3. In contrast to `plotfunc2d`, non-real function values cause an error:

```
>> plotfunc3d(sqrt(1 - x^2 - y^2), x = -1..1, y = -1..1):

 Error: Plot function(s) must return real numbers.
        Type of the returned value is DOM_COMPLEX;
 during evaluation of 'plot3d'
```

Example 4. Piecewise defined functions are handled:

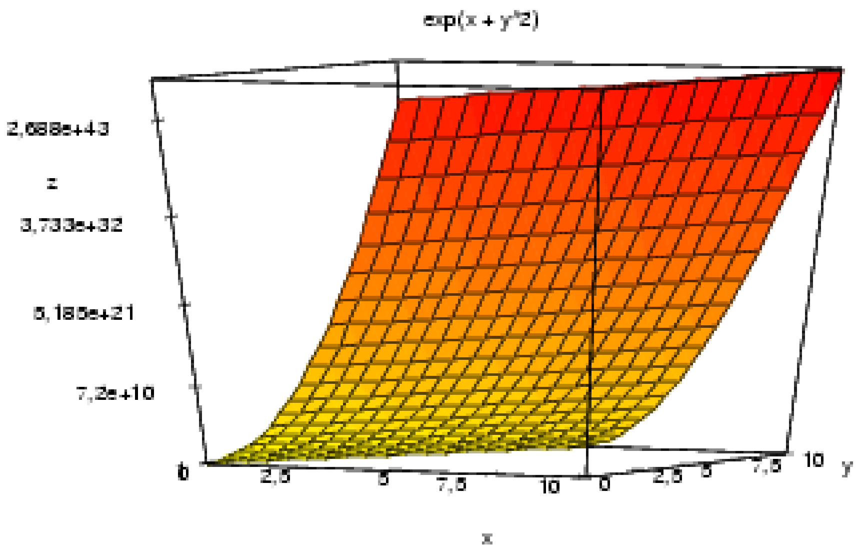
```
>> f := piecewise([x < y, -x^2 + 1], [x >= y, 1 - y^2]):
   plotfunc3d(BackGround = RGB::White,
              ForeGround = RGB::Black,
              Ticks = [Steps = 1, Steps = 1, Steps = 1],
              f(x, y), x = -3..3, y = -3..3)
```




```
>> delete f:
```

Example 5. We use the scene option *AxesScaling* to create a logarithmic plot:

```
>> plotfunc3d(AxesScaling = [Lin, Lin, Log],
              exp(x + y^2), x = 0..10, y = 0..10):
```



---

plotOptions2d – scene options for 2D plots

This page describes the scene options that may be used when generating 2D graphics via plot2d, plotfunc2d, plot::Scene, or plot. Scene options are attributes that determine the general appearance of a graphical scene such as background color, title, axes style etc.

Call(s):

- plot2d(<SceneOpt1, SceneOpt2, ...>, graphical objects)
- plotfunc2d(<SceneOpt1, SceneOpt2, ...>, graphical objects)
- plot::Scene(graphical objects, <SceneOpt1, SceneOpt2, ...>)
- plot(graphical objects, <SceneOpt1, SceneOpt2, ...>)

Related Functions: plot, plot::Scene, plot, plot2d, plotfunc2d, plot3d, plotfunc3d, plotOptions3d

Parameters:

graphical objects — see the help pages of plot2d, plotfunc2d, plot::Scene, and plot for details

Options:

`SceneOpt1, SceneOpt2, ...` — scene options: each is an equation of the form `OptionName = value`.

`OptionName`	admissible values	default value
Arrows	TRUE, FALSE	FALSE
Axes	*Box*, *Corner*, *None*, *Origin*	*Origin*
AxesOrigin	*Automatic*, `[x0, y0]`	*Automatic*
AxesScaling	[*Lin*/*Log*, *Lin*/*Log*]	[*Lin*, *Lin*]
AxesInFront	TRUE, FALSE	FALSE
BackGround	`[r, g, b]`	`RGB::White`
Discont	TRUE, FALSE	FALSE (`plot2d`) TRUE (`plotfunc2d`) FALSE (`plot`) FALSE (`plot::Scene`)
FontFamily	`"helvetica"`, `"lucida"`, ..	`"helvetica"`
FontSize	positive integers	8
FontStyle	`"bold"`, ..	`"bold"`
ForeGround	`[r, g, b]`	`RBG::Black`
GridLines	*Automatic*, *None* or `[xValue, yValue]`. Admissible values for `xValue`, `yValue` are *Automatic*, integers, *Steps* = `d` or *Steps* = `[d, n]`.	*None*
GridLinesColor	`[r, g, b]`	`RGB::Gray`
GridLinesWidth	positive integers	5
GridLinesStyle	*SolidLines*, *DashedLines*	*DashedLines*
Labeling	TRUE, FALSE	TRUE
Labels	`[string, string]`	`["x", "y"]`
LineStyle	*SolidLines*, *DashedLines*	*SolidLines*
LineWidth	positive integers	1
PlotDevice	*Screen*, `"filename"`, `["filename",`*Ascii*`]`, `["filename",`*Binary*`]`	*Screen*
PointStyle	*Circles*, *FilledCircles*, *FilledSquares*, *Squares*	*FilledSquares*
PointWidth	positive integers	30
RealValuesOnly	TRUE, FALSE	FALSE (`plot2d`) TRUE (`plotfunc2d`) FALSE (`plot::Scene`) FALSE (`plot`)
Scaling	*Constrained*, *UnConstrained*	*UnConstrained*

OptionName	admissible values	default value
Ticks	*Automatic*, *None*, an integer or `[xValue, yValue]`. Admissible values for `xValue`, `yValue` are *Automatic*, an integer, *Steps* = `d`, *Steps* = `[d, n]`, or a list of user defined ticks.	*Automatic*
Title	strings	`""` (`plot2d`) `"f(x)"` (`plotfunc2d`) `""` (`plot::Scene`) `""` (`plot`)
TitlePosition	*Above*, *Below*, `[x, y]`	*Above*
ViewingBox	*Automatic* or `[xValue, yValue]`. Admissible values for `xValue`, `yValue` are *Automatic* or a range `a..b`.	*Automatic*

Option <`Arrows = value`>:

This option determines, whether the axes are drawn with or without an arrow tip. Admissible values are `TRUE` or `FALSE`; the default is *Arrows* = `FALSE`. This option is ignored if *Axes* = *None* or *Axes* = *Box*.

Option <`Axes = value`>:

This option sets the style of the axes. Admissible values are *Box*, *Corner*, *None*, and *Origin*; the default is *Axes* = *Origin*.

- With *Axes* = *Box*, a frame around the scene is drawn.
- With *Axes* = *Corner*, the x-axis is drawn below the scene, the y-axis is drawn left of the scene. The axes cross at the lower left corner of the scene.
- With *Axes* = *None*, no axes are drawn.
- With *Axes* = *Origin*, a coordinate cross is drawn. It is centered at the point set by *AxesOrigin*.

Option <`AxesOrigin = value`>:

This option sets the point where the coordinate axes cross. Admissible values are *Automatic* and `[x0, y0]`; the default is *AxesOrigin* = *Automatic*.

- With *AxesOrigin* = *Automatic*, the coordinate axes cross in the mathematical origin $(0, 0)$ of the `x`-`y`-plane, provided it is inside the viewing range of the plot. If this not the case, then the axes cross at the point of the viewing range that is closest to the mathematical origin.
- With *AxesOrigin* = `[x0, y0]`, the coordinate axes cross at the specified point. Admissible values for the coordinates are real numerical expressions as well as the identifiers *XMin*, *XMax*, *YMin*, *YMax*. These are the extremal coordinates of the scene which are determined internally when the plot is evaluated.

Option <`AxesScaling = [xScale, yScale]` >**:**

- This option sets the scaling of the coordinates. Admissible values for `xScale` and `yScale` are either *Lin* for a linear scale or *Log* for a logarithmic scale. The default is *AxesScaling* = [*Lin*,*Lin*].
- For logarithmic scales, the viewing range of the plot should not extend to negative coordinate values. With *RealValuesOnly* = `TRUE`, negative coordinate ranges are clipped from logarithmic plots. With *RealValuesOnly* = `FALSE`, such values cause an error.

Option <`AxesInFront = value`>**:**

- This option determines, whether the axes are drawn in front of the graphical objects of the scene instead of behind. Admissible values are `TRUE` or `FALSE`; the default is *AxesInFront* = `FALSE`. This option is ignored if *Axes* = *None* or *Axes* = *Box*.

Option <`BackGround = [r, g, b]`>**:**

- This option defines the background color, i.e., the color of the canvas. The values `r`, `g`, `b` must be real numbers between `0` and `1`. They represent the red, green, and blue contributions according to the RGB color model. Pre-defined colors are provided by MuPAD's `RGB` data structure. The default is *BackGround* = `[1, 1, 1]` = `RGB::White`.

Option <`Discont = value`>**:**

- This option determines, whether the graphical objects are checked for discontinuities. Admissible values are `TRUE` and `FALSE`; the default is *`Discont`* = `FALSE` for `plot2d`, `plot::Scene`, `plot`, and *`Discont`* = `TRUE` in `plotfunc2d`, respectively.

  - *`Discont`* = `TRUE` enables symbolic checking of discontinuities. If found, unwanted graphical effects such as spurious lines at the discontinuities are eliminated.

  - *`Discont`* = `FALSE` disables the check.

- The symbolic search for discontinuities may be costly. Do specify *`Discont`* = `FALSE` when the objects are known to be continuous!

- Note that some objects of the `plot` library also have an *object* attribute *`Discont`* which overrides the value of the *scene* option *`Discont`*. In particular, for `plot::Function2d` and `plot::Curve2d`, the default of the object attribute is *`Discont`* `= TRUE`, which overrides the default scene option *`Discont`* `= FALSE` in calls to `plot::Scene` and `plot`.

Option <`FontFamily = FontFamilyString` >**:**

- This option defines the font family used for titles, axes labels, and tick labels. The string `FontFamilyString` may be one of `"helvetica"`, `"lucida"` etc. The default is *`FontFamily`* = `"helvetica"`.

Option <`FontSize = n`>**:**

- This option defines the size of the font used for titles, axes labels, and tick labels. The integer `n` may have values between 7 and 36. The default is *`FontSize`* = 8.

Option <`FontStyle = FontStyleString` >**:**

- This option defines the style of the font used for titles, axes labels, and tick labels. The string `FontStyleString` may be one of `"bold"`, ... The default is *`FontStyle`* = `"bold"`.

Option <`ForeGround = [r, g, b]`>**:**

- This option defines the foreground color, i.e., the color for the axes, the axes labels, the tick marks, the tick labels, and the titles. Points and borderlines of filled polygons are also displayed in this color. The values `r`, `g`, `b` must be real numbers between `0` and `1`. They represent the red, green, and blue contributions according to the RGB color model. Predefined colors are provided by MuPAD's `RGB` data structure. The default is *`ForeGround`* = `[0, 0, 0]` = `RGB::Black`.

- Note that the foreground color does not determine the color of the graphical objects. These are either chosen automatically, or they may be defined by the color option of the objects.

Option <`GridLines = value`>**:**

- This option determines whether grid lines are drawn in the background of the plot. Admissible values are *`None`*, *`Automatic`* or a list `[xValue, yValue]`; the default is *`GridLines`* = *`None`*.

  - With *`GridLines`* = *`None`*, no grid lines are drawn.
  - With *`GridLines`* = *`Automatic`*, grid lines are drawn that are attached to the tick marks.
  - With *`GridLines`* = `[xValue, yValue]` the grid lines can be specified separately for each direction.

  The values `xValue` and `yValue` may be *`Automatic`*, a nonnegative integer, *`Steps`* = `d` or *`Steps`* = `[d, n]`.

  - *`Automatic`* produces grid lines attached to the tick marks. *`GridLines`* = [*`Automatic`*, *`Automatic`*] is the same as *`GridLines`* = *`Automatic`*.
  - A nonnegative integer value sets the minimal number of grid lines. The actual number of grid lines as well as their positions are chosen heuristically. If the number 0 is specified, then no grid lines are produced. *`GridLines`* = `n` is equivalent to *`GridLines`* = `[n, n]`.
  - *`Steps`* = `d` produces grid lines at the positions $j\,d$ with all integer values j leading to gridlines inside the viewing range of the plot. The distance `d` between two grid lines must be a real positive value.
  - *`Steps`* = `[d, n]` is equivalent to *`Steps`* = `d/(n+1)`, i.e., further `n` grid lines are placed between the grid lines produced by *`Steps`* = `d`. The parameter `n` must be a nonnegative integer.

Option <`GridLinesColor = [r, g, b]`>**:**

- This option defines the color of the grid lines. The values `r`, `g`, `b` must be real numbers between `0` and `1`. They represent the red, green, and blue contributions according to the RGB color model. Pre-defined colors are provided by MuPAD's `RGB` data structure. The default is *`GridLines`* = `RGB::Gray`.

Option <`GridLinesWidth = n`>**:**

- This option sets the width of the grid lines. Admissible values for `n` are nonnegative integers; the default is *`GridLinesWidth`* = 5.

Option <`GridLinesStyle = value`>**:**

- This option sets the style of the grid lines. Admissible values are *`SolidLines`* and *`DashedLines`*. The default is *`GridLinesStyle`* = *`DashedLines`*.

Option <`Labeling = value`>**:**

- This option determines, whether the axes are displayed with axes labels and tick mark labels. Admissible values are `TRUE` or `FALSE`; the default is *`Labeling`* = `TRUE`.
- Note that the default labeling of the axes may be changed via *`Labels`*. Further, the tick marks and the tick mark labels may be set via *`Ticks`*.

Option <`Labels = [xString, yString]`>**:**

- This option sets the labels of the axes to the text given by the strings `xString` and `yString`. The default is *`Labels`* = `["x", "y"]`.

Option <`LineStyle = value`>**:**

- This option sets the style in which all line objects of the scene are displayed. Admissible values are *`SolidLines`* and *`DashedLines`*; the default is *`LineStyle`* = *`SolidLines`*.

- Line objects are graphs of functions, curves defined by `[Mode = Curve, ..]`, and polygons generated via `[Mode = List, [..polygons..]]`. You can use the option *`LineStyle`* in the graphical objects to override this scene option and display each line object in its individual line style.

Option <`LineWidth = n`>**:**

- This option sets the width of all line objects in the scene. Admissible values for `n` are nonnegative integers; the default is *`LineWidth`* = 1.

- Line objects are graphs of functions, curves defined by `[Mode = Curve, ..]`, and polygons generated via `[Mode = List, [..polygons..]]`. You can use the option *`LineWidth`* in the graphical objects to override this scene option and display each line object in its individual line style.

Option <`PlotDevice = value`>**:**

- This option determines, which plotting device is to be used for rendering the scene. Admissible values are *`Screen`*, a string `"filename"`, `["filename",` *`Ascii`*`]` or `["filename",` *`Binary`*`]`. The default is *`PlotDevice`* = *`Screen`*.

  - With *`PlotDevice`* = *`Screen`*, the plot is displayed on the screen.
  - With *`PlotDevice`* = `["filename", format]`, the plot is written to the file named `filename` in the specified graphical format. Available formats are *`Ascii`* and *`Binary`*. These are mupad specific formats understood by MuPAD's graphical tool VCam. A file in such a format can later be opened and rendered by VCam.
  - *`PlotDevice`* = `"filename"` is the same as *`PlotDevice`* = `["filename",` *`Binary`*`]`.

- Note that MuPAD graphics can also be saved in a variety of standard graphical formats such as *`Postscript`*, *JPEG*, *TIFF* etc. However, conversion into these formats cannot be specified by a plot command inside a MuPAD session. You have to use the graphical interface of the rendering tool VCam: In a MuPAD Pro notebook, double click on the graphics to activate the VCam interface. Using the menu item "Edit/Save Graphics...", you can choose the desired format in the "Export Graphics" dialog box.

Option <`PointStyle = value`>**:**

- This option sets the style in which all point objects in the current scene are displayed. Admissible values are *Circles*, *Squares*, *FilledCircles*, and *FilledSquares*. The default is *PointStyle* = *FilledSquares*.
- Point objects are graphical primitives generated via MuPAD's function `point`. They can be displayed via `plot2d` using objects of the type `[Mode = List, [..points..]]`. You can use the object option *PointStyle* to override this scene option and display each point with its individual style.

Option <`PointWidth = n`>**:**

- This option sets the size of all point objects in the current scene. Admissible values for `n` are positive integers; the default is *PointWidth* = 30.
- Point objects are graphical primitives generated via MuPAD's function `point`. They can be displayed via `plot2d` using objects of the type `[Mode = List, [..points..]]`. You can use the object option *PointWidth* to override this scene option and display each point with its individual width.

Option <`RealValuesOnly = value`>**:**

- If a graphical object such as a function produces a complex value during the evaluation of the plot, then an error occurs. Specifying *RealValuesOnly* = `TRUE`, such errors are trapped. Only those parts of the objects producing real values are plotted. E.g., with this option the function `sqrt(x)` can be plotted over the interval $x \in [-1, 1]$: the plot only displays the real function values for $x \geq 0$.

  With *RealValuesOnly* = `FALSE` no internal check is performed. The renderer produces an error, when it encounters a complex value.

  The default is *RealValuesOnly* = `FALSE` in `plot2d`, `plot::Scene`, and `plot`, while it is *RealValuesOnly* = `TRUE` in `plotfunc2d`.
- The short form *RealsOnly* is synonymous with *RealValuesOnly*.
- Checking for real values may be costly. Do specify *RealsOnly* = `FALSE` when the objects are known to be real valued!

⌘ Note that some objects of the `plot` library also have an *object* attribute ***RealValuesOnly*** which overrides the value of the *scene* option ***RealValuesOnly***. In particular, for `plot::Function2d` and `plot::Curve2d`, the default of the object attribute is ***RealValuesOnly*** `= TRUE`, which overrides the default scene option ***RealValuesOnly*** `= FALSE` in calls to `plot::Scene` and `plot`.

Option <`Scaling = value`>**:**

⌘ This option determines the aspect ratio of the x and y coordinates. Admissible values are ***Constrained*** and ***UnConstrained***; the default is ***Scaling*** = ***UnConstrained***.

- With ***Scaling*** = ***Constrained***, the aspect ratio of the coordinates is 1 : 1. In particular, circles appear as circles. This mode is not appropriate, if the x-diameter of the scene differs significantly from the y-diameter.
- With ***Scaling*** = ***UnConstrained***, the aspect ratio of the coordinates is chosen such that the scene fills the canvas optimally. In particular, circles may appear as ellipses.

Option <`Ticks = value`>**:**

⌘ This option defines the ticks on the axes. Admissible values are ***None***, ***Automatic***, a nonnegative integer or a list `[xValue, yValue]`. The default is ***Ticks*** = ***Automatic***.

- With ***Ticks*** = ***None***, no ticks are drawn.
- With ***Ticks*** = ***Automatic***, ticks are chosen heuristically.
- With ***Ticks*** = `n`, the minimum value for the ticks on both axes is specified by the nonnegative integer `n`. Note that more ticks than specified may be drawn in order to place them at reasonable positions.
- With ***Ticks*** = `[xValue, yValue]`, the ticks can be specified separately for each axis.

The values `xValue` and `yValue` may be ***Automatic***, a nonnegative integer, ***Steps*** = `d`, ***Steps*** = `[d, n]`, or a list of user-defined ticks.

- ***Automatic*** produces heuristically chosen ticks. ***Ticks*** = ***Automatic*** is equivalent to ***Ticks*** = [***Automatic***, ***Automatic***].

- A nonnegative integer value sets the minimal number of ticks. The actual number of ticks as well as their positions are chosen heuristically. If the number 0 is specified, then no tick marks are produced. *Ticks* = `n` is equivalent to *Ticks* = `[n, n]`.

- *Steps* = `d` produces ticks at the positions $j\,d$ with all integer values j leading to ticks inside the viewing range of the plot. The distance `d` between two ticks must be a real positive value.

- *Steps* = `[d, n]` produces the same "large" ticks as *Steps* = `d`. Between such ticks further `n` smaller ticks are positioned. The parameter `n` must be a nonnegative integer. The "large" ticks carry labels if *Labeling* = `TRUE`. The "small" ticks do not carry labels.

- Ticks can be placed at arbitrary positions by a list `[t1, t2, ..]`. Admissible values for `t1`, `t2` etc. are real numerical expressions defining the positions of the ticks. Alternatively, any element of the list may be an equation of the form `t = label`, where `t` is a numerical value and `label` is a string. This produces a tick at the position `t` with the string as label. The label is diplayed if *Labeling* = `TRUE` is specified. E.g.,

  Ticks `= [[0.2, PI = "PI"], [sqrt(2), 2, 3]]`

  produces two ticks on the x-axis at the positions $x = 0.2$ and $x = \pi$. The second tick carries the label `"PI"`. On the y-axis, three ticks without lables are produced.

  If ticks outside the viewing range of the plot are specified, then the viewing range is extended automatically such that all ticks are visible.

Option <`Title = TitleString` >**:**

This option adds the text given by the string `TitleString` to the scene. In `plot2d`, `plot::Scene`, and `plot`, the default is the empty string *Title* = `""`, i.e., no title. In `plotfunc2d`, the expressions defining the functions to be plotted are converted to title strings.

Option <`TitlePosition = value`>**:**

This option determines the position of the title. Admissible values are *Above*, *Below*, and `[x, y]`; the default is *TitlePosition* = *Above*.

- With *TitlePosition* = *Above*, the title is centered above the scene.

- With *TitlePosition* = *Below*, the title is centered below the scene.

- With *TitlePosition* = [x, y], the title may be placed at any position in the scene. The parameters x, y must be real numerical values between 0 and 10. The position [0, 0] denotes the upper left corner of the scene, the position [10, 10] denotes the lower right corner.

- The title can be dragged interactively with the mouse to any appropriate position inside the scene.

Option <ViewingBox = value>:

- This option sets the viewing box for the scene, i.e., the range of x and y coordinates that are visible on the canvas. Admissible values are *Automatic* and [xValue, yValue]; the default is *ViewingBox* = *Automatic*.

  - With *ViewingBox* = *Automatic*, the viewing box is chosen such that the entire scene is visible.

  - The values xValue and yValue may be *Automatic* or a range a..b. Admissible values for a and b are real numerical expressions as well as the identifiers *XMin*, *XMax*, *YMin*, *YMax*. These are the extremal coordinates of the scene which are determined internally when the plot is evaluated.

- Clipping to a viewing box can be expensive! Do use the default *ViewingBox* = *Automatic* whenever this is appropriate.

Changes:

- The new option *AxesInFront* was introduced.

---

plotOptions3d – scene options for 3D plots

This page describes the scene options that may be used when generating 3D graphics via plot3d, plotfunc3d, plot::Scene, or plot. Scene options are attributes that determine the general appearance of a graphical scene such as camera point, background color, title, axes style etc.

Call(s):

- plot3d(<SceneOpt1, SceneOpt2, ...>, graphical objects)
- plotfunc3d(<SceneOpt1, SceneOpt2, ...>, graphical objects)
- plot::Scene(graphical objects, <SceneOpt1, SceneOpt2, ...>)
- plot(graphical objects, <SceneOpt1, SceneOpt2, ...>)

Parameters:

graphical objects — see the help pages of plot3d, plotfunc3d, plot::Scene, and plot for details

Options:

SceneOpt1, SceneOpt2, .. — scene options: each is an equation of the form OptionName = value.

OptionName	admissible values	default value
Arrows	TRUE, FALSE	FALSE
Axes	*Box*, *Corner*, *None*, *Origin*	*Box*
AxesOrigin	*Automatic*, [x0, y0, z0]	*Automatic*
AxesScaling	[*Lin*/*Log*, *Lin*/*Log*, *Lin*/*Log*]	[*Lin*, *Lin*, *Lin*]
BackGround	[r, g, b]	RGB::White
CameraPoint	*Automatic*, [x, y, z]	*Automatic*
FocalPoint	*Automatic*, [x, y, z]	*Automatic*
FontFamily	"helvetica", "lucida", ..	"helvetica"
FontSize	positive integers	8
FontStyle	"bold", ..	"bold"
ForeGround	[r, g, b]	RBG::Black
Labeling	TRUE, FALSE	TRUE
Labels	[string, string, string]	["x","y","z"]
LineStyle	*SolidLines*, *DashedLines*	*SolidLines*
LineWidth	positive integers	1
PlotDevice	*Screen*, "filename", ["filename",*Ascii*], ["filename",*Binary*]	*Screen*
PointStyle	*Circles*, *FilledCircles*, *FilledSquares*, *Squares*	*FilledSquares*
PointWidth	positive integers	30
Scaling	*Constrained*, *UnConstrained*	*UnConstrained*
Ticks	*Automatic*, *None*, an integer or [xValue, yValue, zValue]. Admissible values for xValue, yValue, zValue are *Automatic*, an integer, *Steps* = d, *Steps* = [d, n], or a list of user defined ticks.	*Automatic*

`OptionName`	admissible values	default value
Title	strings	`""` (`plot3d`) `"f(x, y)"` (`plotfunc3d`) `""` (`plot::Scene`) `""` (`plot`)
TitlePosition	*Above*, *Below*, `[x, y]`	*Above*
ViewingBox	*Automatic*	*Automatic*

Related Functions: `plot`, `plot::Scene`, `plot`, `plot2d`, `plotfunc2d`, `plotOptions2d`, `plot3d`, `plotfunc3d`

Option <`Arrows = value`>**:**

This option determines, whether the axes are drawn with or without an arrow tip. Admissible values are `TRUE` or `FALSE`; the default is *Arrows* = `FALSE`. This option is ignored if *Axes* = *None* or *Axes* = *Box*.

Option <`Axes = value`>**:**

This option sets the style of the axes. Admissible values are *Box*, *Corner*, *None*, and *Origin*; the default is *Axes* = *Box*.

- With *Axes* = *None*, no axes are drawn.
- With *Axes* = *Box*, a box around the scene is drawn.
- With *Axes* = *Corner*, a coordinate cross is drawn. It is centered at one of the corners of the scene.
- With *Axes* = *Origin*, a coordinate cross is drawn. It is centered at the point set by *AxesOrigin*.

Option <`AxesOrigin = value`>**:**

This option sets the point where the coordinate axes cross. Admissible values are *Automatic* and `[x0, y0, z0]`; the default is *AxesOrigin* = *Automatic*.

- With *AxesOrigin* = *Automatic*, the coordinate axes cross in the mathematical origin $(0, 0, 0)$, provided it is inside the viewing range of the plot. If this not the case, then the axes cross at the point of the viewing range that is closest to the mathematical origin.

- With *AxesOrigin* = [x0, y0, z0], the coordinate axes cross at the specified point. Admissible values for the coordinates are real numerical expressions as well as the identifiers *XMin*, *XMax*, *YMin*, *YMax*, *ZMin*, *ZMax*. These are the extremal coordinates of the scene which are determined internally when the plot is evaluated.

Option <`AxesScaling = [xScale, yScale, zScale]` >**:**

- This option sets the scaling of the coordinates. Admissible values for `xScale`, `yScale`, and `zScale` are either *Lin* for a linear scale or *Log* for a logarithmic scale. The default is *AxesScaling* = [*Lin* ,*Lin* ,*Lin*].

- For logarithmic scales, make sure that the viewing range of the plot does not extend to negative coordinate values. Otherwise, an error occurs!

Option <`BackGround = [r, g, b]`>**:**

- This option defines the background color, i.e., the color of the canvas. The values `r`, `g`, `b` must be real numbers between `0` and `1`. They represent the red, green, and blue contributions according to the RGB color model. Pre-defined colors are provided by MuPAD's `RGB` data structure. The default is *BackGround* = `[1, 1, 1]` = `RGB::White`.

Option <`CameraPoint = value`>**:**

- This option sets the position of the observer's camera. The optical axis of the camera is the vector from the *CameraPoint* to the *FocalPoint*. The value may be *Automatic* or a vector `[x, y, z]`. With the default *Automatic*, the camera position is chosen automatically outside the graphical scene. Also a user-defined point should lie outside the scene. A point close to the scene leads to perspective distortion. A point far from the scene prevents such distortion (parallel projection). The size of the projected scene is independent of the distance: the projection is enlarged automatically such that it fills the canvas.

Option <`FocalPoint = value`>**:**

- This option sets the point the observer's camera is pointing to. The optical axis of the camera is the vector from the *CameraPoint* to the *FocalPoint*. The value may be *Automatic* or a vector `[x, y, z]`. With the default *Automatic*, the focal point is chosen automatically as the center of the graphical scene.

Option <`FontFamily = FontFamilyString` >**:**

- This option defines the font family used for titles, axes labels, and tick labels. The string `FontFamilyString` may be one of `"helvetica"`, `"lucida"` etc. The default is *FontFamily* = `"helvetica"`.

Option <`FontSize = n`>**:**

- This option defines the size of the font used for titles, axes labels, and tick labels. The integer `n` may have values between 7 and 36. The default is *FontSize* = 8.

Option <`FontStyle = FontStyleString` >**:**

- This option defines the style of the font used for titles, axes labels, and tick labels. The string `FontStyleString` may be one of `"bold"`, ... The default is *FontStyle* = `"bold"`.

Option <`ForeGround = [r, g, b]`>**:**

- This option defines the foreground color, i.e., the color for the axes, the axes labels, the tick marks, the tick labels, and the titles. Points and borderlines of filled polygons are also displayed in this color. The values `r`, `g`, `b` must be real numbers between `0` and `1`. They represent the red, green, and blue contributions according to the RGB color model. Predefined colors are provided by MuPAD's `RGB` data structure. The default is *ForeGround* = `[0, 0, 0]` = `RGB::Black`.
- Note that the foreground color does not determine the color of the graphical objects. These are either chosen automatically, or they may be defined by the color option of the objects.

Option <`Labeling = value`>**:**

- This option determines, whether the axes are displayed with axes labels and tick mark labels. Admissible values are `TRUE` or `FALSE`; the default is *Labeling* = `TRUE`.

⌘ Note that the default labeling of the axes may be changed via *Labels*. Further, the tick marks, and the tick mark labels may be set via *Ticks*.

Option <`Labels = [xString, yString, zString]`>**:**

⌘ This option sets the labels of the axes to the text given by the strings `xString`, `yString`, and `zString`. The default is *Labels* = `["x", "y", "z"]`.

Option <`LineStyle = value`>**:**

⌘ This option sets the style in which all line objects of the scene are displayed. Admissible values are *SolidLines* and *DashedLines*; the default is *LineStyle* = *SolidLines*.

⌘ Line objects are graphs of functions, curves defined by `[Mode = Curve, ..]`, the parameter lines of surfaces, and polygons generated via `[Mode = List, [..polygons..]]`. You can use the option *LineStyle* in the graphical objects to override this scene option and display each line object in its individual line style.

Option <`LineWidth = n`>**:**

⌘ This option sets the width of all line objects in the scene. Admissible values for `n` are nonnegative integers; the default is *LineWidth* = 1.

⌘ Line objects are graphs of functions, curves defined by `[Mode = Curve, ..]`, and polygons generated via `[Mode = List, [..polygons..]]`. You can use the option *LineWidth* in the graphical objects to override this scene option and display each line object in its individual line style.

Option <`PlotDevice = value`>**:**

⌘ This option determines, which plotting device is to be used for rendering the scene. Admissible values are *Screen*, a string `"filename"`, `["filename", `*Ascii*`]` or `["filename", `*Binary*`]`. The default is *PlotDevice* = *Screen*.

- With *PlotDevice* = *Screen*, the plot is displayed on the screen.

- With *PlotDevice* = `["filename", format]`, the plot is written to the file named `filename` in the specified graphical formats. Available formats are *Ascii* and *Binary*. These are mupad specific formats understood by MuPAD's graphical tool VCam. A file in such a format can later be opened and rendered by VCam.

- *PlotDevice* = `"filename"` is the same as *PlotDevice* = `["filename",` *Binary*`]`.

⊞ Note that MuPAD graphics can also be saved in a variety of standard graphical formats such as *Postscript*, *JPEG*, *TIFF* etc. However, conversion into these formats cannot be specified by a plot command inside a MuPAD session. You have to use the graphical interface of the rendering tool VCam: In a MuPAD Pro notebook, double click on the graphics to activate the VCam interface. Using the menu item "Edit/Save Graphics ..", you can choose the desired format in the "Export Graphics" dialog box.

Option <`PointStyle = value`>**:**

⊞ This option sets the style in which all point objects in the current scene are displayed. Admissible values are *Circles*, *Squares*, *FilledCircles*, and *FilledSquares*. The default is *PointStyle* = *FilledSquares*.

⊞ Point objects are graphical primitives generated via MuPAD's function `point`. They can be displayed via `plot3d` using objects of the type `[Mode = List, [..points..]]`. You can use the object option *PointStyle* to override this scene option and display each point with its individual style.

Option <`PointWidth = n`>**:**

⊞ This option sets the size of all point objects in the current scene. Admissible values for `n` are positive integers; the default is *PointWidth* = 30.

⊞ Point objects are graphical primitives generated via MuPAD's function `point`. They can be displayed via `plot3d` using objects of the type `[Mode = List, [..points..]]`. You can use the object option *PointWidth* to override this scene option and display each point with its individual width.

Option <`Scaling = value`>:

This option determines the aspect ratio of the x, y, z coordinates. Admissible values are *`Constrained`* and *`UnConstrained`*; the default is *`Scaling`* = *`UnConstrained`*.

- With *`Scaling`* = *`Constrained`*, the aspect ratio of the coordinates is 1 : 1 : 1. In particular, spheres appear as spheres. This mode is not appropriate if the diameters of the scene in the three directions differ significantly.
- With *`Scaling`* = *`UnConstrained`*, the aspect ratio of the coordinates is chosen such that the scene fills the canvas optimally. In particular, spheres may appear as ellipsoids.

Option <`Ticks = value`>:

This option defines the ticks on the axes. Admissible values are *`None`*, *`Automatic`*, a nonnegative integer or a list `[xValue, yValue, zValue]`. The default is *`Ticks`* = *`Automatic`*.

- With *`Ticks`* = *`None`*, no ticks are drawn.
- With *`Ticks`* = *`Automatic`*, ticks are chosen heuristically.
- With *`Ticks`* = `n`, the minimum value for the ticks on the three axes is specified by the nonnegative integer `n`. Note that more ticks than specified may be drawn in order to place them at reasonable positions.
- With *`Ticks`* = `[xValue, yValue, zValue]`, the ticks can be specified separately for each axis.

The values `xValue`, `yValue`, and `zValue` may be *`Automatic`*, a nonnegative integer, *`Steps`* = `d`, *`Steps`* = `[d, n]`, or a list of user-defined ticks.

- *`Automatic`* produces heuristically chosen ticks. *`Ticks`* = *`Automatic`* is equivalent to *`Ticks`* = `[`*`Automatic`*`,` *`Automatic`*`,` *`Automatic`*`]`.
- A nonnegative integer value sets the minimal number of ticks. The actual number of ticks as well as their positions are chosen heuristically. If the number 0 is specified, then no tick marks are produced. *`Ticks`* = `n` is equivalent to *`Ticks`* = `[n, n, n]`.
- *`Steps`* = `d` produces ticks at the positions $j\,d$ with all integer values j leading to ticks inside the viewing range of the plot. The distance `d` between two ticks must be a real positive value.

- *Steps* = [d, n] produces the same "large" ticks as *Steps* = d. Between such ticks further n smaller ticks are positioned. The parameter n must be a nonnegative integer. The "large" ticks carry labels if *Labeling* = TRUE. The "small" ticks do not carry labels.

- Ticks can be placed at arbitrary positions by a list [t1, t2, ..]. Admissible values for t1, t2 etc. are real numerical expressions defining the positions of the ticks. Alternatively, any element of the list may be an equation of the form t = label, where t is a numerical value and label is a string. This produces a tick at the position t with the string as label. The label is diplayed if *Labeling* = TRUE is specified. E.g.,

  Ticks = [[0.2, PI = "PI"], [sqrt(2), 2, 3], [1, 2, 3]]

  produces two ticks on the x-axis at the positions $x = 0.2$ and $x = \pi$. The second tick carries the label "PI". On the y- and z-axes, three ticks without lables are produced.

  If ticks outside the viewing range of the plot are specified, then the viewing range is extended automatically such that all ticks are visible.

Option <Title = TitleString >**:**

This option adds the text given by the string TitleString to the scene. In plot3d, plot::Scene, and plot, the default is the empty string *Title* = "", i.e., no title. In plotfunc3d, the expressions defining the functions to be plotted are converted to title strings.

Option <TitlePosition = value>**:**

This option determines the position of the title. Admissible values are *Above*, *Below*, and [x, y]; the default is *TitlePosition* = *Above*.

- With *TitlePosition* = *Above*, the title is centered above the scene.
- With *TitlePosition* = *Below*, the title is centered below the scene.
- With *TitlePosition* = [x, y], the title may be placed at any position in the scene. The parameters x, y must be real numerical values between 0 and 10. The position [0, 0] denotes the upper left corner of the scene, the position [10, 10] denotes the lower right corner.

The title can be dragged interactively with the mouse to any appropriate position inside the scene.

Option <`ViewingBox = value`>:

- This option sets the viewing box for the scene, i.e., the range of x, y, z that are visible on the canvas. Presently, the only admissible value is ***`Automatic`***. The viewing box is chosen such that the entire scene is visible.

`point` – generate a graphical point primitive

`point(x, y)` defines a 2D point with the coordinates `x` and `y`.

`point(x, y, z)` defines a 3D point with the coordinates `x`, `y` and `z`.

Call(s):

- `point(x, y` <`,` *`Color`* `= [r, g, b]`>`)`
- `point(x, y, z` <`,` *`Color`* `= [r, g, b]`>`)`

Parameters:

`x, y, z` — real numbers

Options:

`Color` `= [r, g, b]` — sets an RGB color given by the amount of red, green, and blue. The parameters `r`, `g`, `b` must be real numbers between 0 and 1.

Return Value: an object of type `DOM_POINT`.

Related Functions: `plot`, `plot::Point`, `plot2d`, `plot3d`, `plotfunc2d`, `plotfunc3d`, `polygon`, `RGB`

Details:

- `point` defines a 2D or 3D point. It can be displayed graphically via `plot2d`/`plot3d` using the list format `[Mode = List, [..points..]]`.
- NOTE: The coordinates and color values must be numerical expressions that can be converted to real floating point numbers. Symbolic expressions such as `PI + 1`, `exp(sqrt(2))` etc. are accepted and converted to floating point numbers automatically. Note, however, that expressions involving symbolic identifiers are not accepted! Cf. example 3.

- The plot library provides the alternative point primitive `plot::Point`. This object is more flexible than the kernel object generated by `point`. The first can be used with all functions of the `plot` library, whereas the latter can only be used in a call to `plot2d` or `plot3d`.

- `point` is a function of the system kernel.

Option <`Color = [r, g, b]`>:

- The color values `r`, `g`, `b` must be numerical expressions that can be converted to real floating point numbers from the interval $[0.0, 1.0]$. An error occurs if any of these values is not in this range. Symbolic expressions such as `PI - 2`, `exp(-sqrt(2))` etc. are accepted. Note, however, that expressions involving symbolic identifiers are not accepted! Cf. example 3.

- The domain `RGB` contains many predefined colors.

Operands: The first two, respectively three, operands of a point are the coordinates. The last operand is the list `[r, g, b]` defining the point color. This operand is `NIL` if no color was specified.

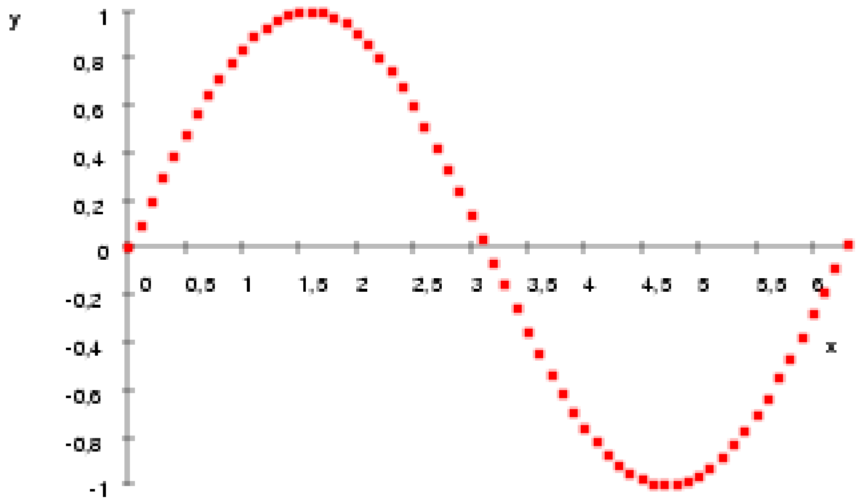
Example 1. `point` with two arguments defines a 2D point:

```
>> point(1, PI)

                    point(1, 3.141592654)
```

Points generated by `point` represent graphical primitives that can be displayed via `plot2d` and `plot3d` using the list format `[Mode = List, [..points..]]`:

```
>> plot2d(Scaling = UnConstrained, PointWidth = 30,
          [Mode = List, [point(i/10, sin(i/10)) $ i=0..63]])
```



Example 2. Points may be defined with a given color:

```
>> point(0, 1, PI, Color = [1/2, 0, PI - 2*sqrt(2)])

   point(0, 1, 3.141592654, Color = [0.5, 0.0, 0.3131655288])
```

The domain RGB contains many pre-defined colors:

```
>> point(1.0, 0.0, 1.0, Color = RGB::Red)

          point(1.0, 0.0, 1.0, Color = [1.0, 0.0, 0.0])
```

Example 3. Symbolic coordinates or colors are not accepted:

```
>> point(x, y, z)

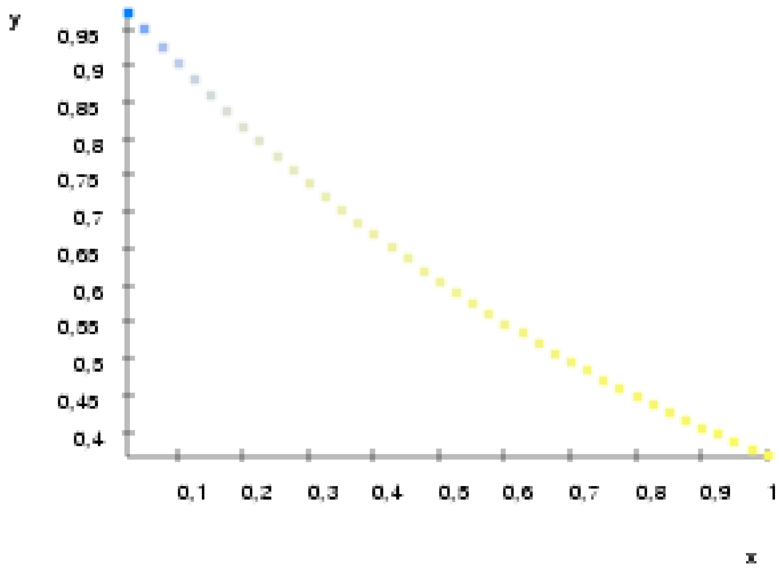
 Error: Illegal argument [point]

>> point(1, 2, Color = [r, g, b])

 Error: Illegal color specification [point]
```

However, one can create lists of points using symbolic loop variables:

```
>> mypoints := [point(i/40, exp(-i/40),
                      Color = [1 - 1/i, i/(1 + i), exp(-i/40)])
                $ i = 1..40]:
   plot2d(PointWidth = 30, [Mode = List, mypoints])
```




```
>> delete mypoints:
```

---

poly – create a polynomial

poly(f) converts a polynomial expression f to a polynomial of the kernel domain DOM_POLY.

Call(s):

- poly(f <, [x1, x2, ...]> <, ring>)
- poly(p <, [x1, x2, ...]> <, ring>)
- poly(list, [x1, x2, ...] <, ring>)

Parameters:

`f`	—	a polynomial expression
`x1, x2, ...`	—	the indeterminates of the polynomial: typically, identifiers or indexed identifiers.
`ring`	—	the coefficient ring: either *Expr*, or *IntMod*(n) with some integer $n > 1$, or a domain of type `DOM_DOMAIN`. The default is the ring *Expr* of arbitrary MuPAD expressions.
`p`	—	a polynomial of type `DOM_POLY` generated by `poly`
`list`	—	a list containing coefficients and exponents

Return Value: a polynomial of the domain type `DOM_POLY`. `FAIL` is returned if conversion to a polynomial is not possible.

Related Functions: `Dom::DistributedPolynomial`, `Dom::MultivariatePolynomial`, `Dom::Polynomial`, `Dom::UnivariatePolynomial`, `RootOf`, `coeff`, `collect`, `degree`, `degreevec`, `divide`, `evalp`, `expr`, `factor`, `gcd`, `ground`, `indets`, `lcoeff`, `ldegree`, `lmonomial`, `lterm`, `mapcoeffs`, `nterms`, `nthcoeff`, `nthmonomial`, `nthterm`, `poly2list`, `polylib`, `tcoeff`

Details:

- MuPAD provides the kernel domain `DOM_POLY` to represent polynomials. The arithmetic for this data structure is more efficient than the arithmetic for polynomial expressions. Moreover, this domain allows to use special coefficient rings that cannot be represented by expressions. The function `poly` is the tool for generating polynomials of this type.

- `poly(f, [x1, x2, ...], ring)` converts the expression `f` to a polynomial in the indeterminates `x1, x2, ...` over the specified coefficient ring. The expression `f` need not be entered in expanded form, it is internally expanded by `poly`.

  If no indeterminates are given, they are searched for internally. An error occurs if no indeterminates are found.

  The ring `Expr` is used if no coefficient ring is specified. In this case, arbitrary MuPAD expressions are allowed as coefficients.

  `poly` returns `FAIL` if the expression cannot be converted to a polynomial. Cf. example 9.

- `poly(p, [x1, x2, ...], ring)` converts a polynomial `p` of type `DOM_POLY` to a polynomial in the indeterminates `x1, x2, ...` over the specified coefficient ring. Note that both the indeterminates as well as the coefficient ring are part of the data structure `DOM_POLY`. This call may be used to change these data in a given polynomial `p` of this type.

  If no indeterminates are specified, the indeterminates of `p` are used.

If no coefficient ring is specified, the ring of `p` is used.

Cf. examples 7 and 8.

- `poly(list, [x1, x2, ...], ring)` converts a list of coefficients and exponents to a polynomial in the indeterminates `x1, x2, ...` over the specified coefficient ring. Cf. examples 3 and 6. This call is the fastest way of creating a polynomial of type `DOM_POLY`.

  The list must contain an element for each non-zero monomial of the polynomial, i.e., it is possible to use sparse input involving only non-zero terms. In particular, an empty list results in the zero polynomial.

  Each element of the list must in turn be a list with two elements: the coefficient of the monomial and the exponent or exponent vector. For a univariate polynomial in the variable x, say, the list

  $$[[c_1, e_1], [c_2, e_2], \ldots]$$

  corresponds to $c_1\,x^{e_1} + c_2\,x^{e_2} + \cdots$. For a multivariate polynomial, the exponent vectors are lists containing the exponents of all indeterminates of the polynomial. The order of the exponents must be the same as the order given by the list of indeterminates. For a multivariate polynomial in the variables $x_1, x_2, \ldots$, say, the term list

  $$[[c_1, [e_{11}, e_{12}, \ldots]], [c_2, [e_{21}, e_{22}, \ldots]], \ldots]$$

  corresponds to $c_1\,x_1^{e_{11}}\,x_2^{e_{12}} \cdots + c_2\,x_1^{e_{21}}\,x_2^{e_{22}} \cdots + \cdots$.

  The order of the elements of the term list does not affect the resulting polynomial. There may be multiple entries corresponding to the same term: the coefficients are added in such cases.

  With this call, term lists returned by `poly2list` can be reconverted to polynomials.

- The order of the indeterminates is given by their position in the input list `[x1, x2, ...]`. If not specified, they are searched for internally in the expression `f` and their order is determined by the system. Cf. example 2.

- The indeterminates need not be identifiers or indexed identifiers. Any expression can be used as an indeterminate as long as it is not rational. E.g., the expressions `sin(x)`, `f(x)`, or `y^(1/3)` are accepted as indeterminates. Cf. example 4.

- `poly` is a function of the system kernel.

Option <`ring`>:

- The default ring ***Expr*** represents arbitrary MuPAD expressions. Mathematically, this ring coincides with `Dom::ExpressionField()`. Note, however, that polynomials distinguish ***Expr*** and `Dom::ExpressionField()`. In particular, arithmetic over ***Expr*** is faster.

- The ring *IntMod*(n) represents the residue class ring Z/nZ, using the symmetrical representation. Here, n must be an integer greater than 1. Mathematically, this ring coincides with Dom::IntegerMod(n). Note, however, that polynomials distinguish *IntMod*(n) and Dom::IntegerMod(n). In particular, arithmetic over *IntMod* is faster, coefficients requested by coeff etc. are returned as integers of type DOM_INT. Cf. examples 5, 6, and 8.

- Any domain of type DOM_DOMAIN can be used as a coefficient ring if the domain provides arithmetical operations. See the "Background" section below for further details.

  If a coefficient domain is specified, only elements of the domain are accepted as coefficients. On input, poly tries to convert a polynomial expression f to a polynomial over the coefficient ring. For some coefficient rings, however, it is not possible to use arithmetical expressions to represent a polynomial, because multiplication with the indeterminates may not be a valid operation in the ring. In this case, the polynomial can be defined via a term list. Cf. example 6.

Example 1. A call of poly creates a polynomial from a polynomial expression:

```
>> p := poly(2*x*(x + 3))

                                 2
                        poly(2 x  + 6 x, [x])
```

The operators *, +, - and ^ work on polynomials:

```
>> p^2 - p + poly(x, [x])

                        4        3        2
                poly(4 x  + 24 x  + 34 x  - 5 x, [x])
```

For multiplication with a constant, one must either convert the constant to a polynomial of the appropriate type, or one can use multcoeffs:

```
>> poly(c, [x])*p = multcoeffs(p, c)

               2                                2
  poly((2 c) x  + (6 c) x, [x]) = poly((2 c) x  + (6 c) x, [x])

>> delete p:
```

Example 2. A polynomial may be created with parameters. In the following call, y is a parameter and not an indeterminate:

```
>> poly((x*(y + 1))^2, [x])

                           2  2
             poly((y + 1)  x , [x])
```

If no indeterminates are specified, they are searched for automatically. In the following call, the previous expression is converted to a multivariate polynomial:

```
>> poly((x*(y + 1))^2)

                2  2         2    2
          poly(y  x  + 2 y x  + x , [y, x])
```

The order of the indeterminates can be specified explicitly:

```
>> poly((x*(y + 1))^2, [x, y])

                2  2       2      2
          poly(x  y  + 2 x  y + x , [x, y])
```

Example 3. The following polynomials are created by term lists:

```
>> poly([[c2, 3], [c1, 7], [c3, 0]], [x])

                     7       3
            poly(c1 x  + c2 x  + c3, [x])

>> poly([[c2, 3], [c1, 7], [c3, 0], [a, 3]], [x])

                  7             3
         poly(c1 x  + (a + c2) x  + c3, [x])
```

For multivariate polynomials, exponent vectors must be specified via lists:

```
>> poly([[c1, [2, 2]], [c2, [2, 1]], [c3, [2, 0]]], [x, y])

                  2  2       2         2
        poly(c1 x  y  + c2 x  y + c3 x , [x, y])
```

Example 4. Expressions such as f(x) may be used as indeterminates:

```
>> poly(f(x)*(f(x) + x^2))

               2            2
         poly(x  f(x) + f(x) , [x, f(x)])
```

Example 5. The residue class ring *IntMod*(7) is a valid coefficient ring:

```
>> p := poly(9*x^3 + 4*x - 7, [x], IntMod(7))

                        3
                poly(2 x  - 3 x, [x], IntMod(7))
```

Internally, modular arithmetic is used when computing with polynomials over this ring:

```
>> p^3

                  9    7      5    3
            poly(x  - x  - 2 x  + x , [x], IntMod(7))
```

Note, however, that coefficients are not returned as elements of a special domain, but as plain integers of type `DOM_INT`:

```
>> coeff(p)

                              2, -3

>> delete p:
```

Example 6. The input syntax using term lists may be combined with a given coefficient ring:

```
>> poly([[9, 3], [4, 1], [-2, 0]], [x], IntMod(7))

                      3
              poly(2 x  - 3 x - 2, [x], IntMod(7))
```

Note that the input coefficients are interpreted as elements of the coefficient domain, i.e., conversions such as 9 mod 7 $\rightarrow$ 2 occur on input. We can also use the domain `Dom::IntegerMod(7)` to define an equivalent polynomial. However, in contrast to *IntMod*(7), the coefficients a represented by the numbers $0, \ldots, 6$ rather than $-3, \ldots, 3$:

```
>> poly([[9, 3], [4, 1], [-2, 0]], [x], Dom::IntegerMod(7))

                  3
          poly(2 x  + 4 x + 5, [x], Dom::IntegerMod(7))
```

Note that the following attempt to define a polynomial via an expression fails, because the domain `Dom::IntegerMod(7)` does not permit multiplication with identifiers:

```
>> c := Dom::IntegerMod(7)(3)
```

```
                 3 mod 7
```

```
>> poly(c*x^2, [x], Dom::IntegerMod(7))
```

```
                  FAIL
```

In such a case, term lists allow to specify the polynomial:

```
>> poly([[c, 2]], [x], Dom::IntegerMod(7))

                         2
             poly(3 x , [x], Dom::IntegerMod(7))
```

```
>> delete c:
```

Example 7. It is possible to change the indeterminates in a polynomial:

```
>> p:= poly(((a + b)*x - a^2)*x, [x]): p, poly(p, [a, b])

              2       2
 poly((a + b) x  + (- a ) x, [x]),

              2    2      2
    poly((-x) a  + x  a + x  b, [a, b])
```

Example 8. It is possible to change the coefficient ring of a polynomial:

```
>> p := poly(-4*x + 5*y - 5, [x, y], IntMod(7)):
   p, poly(p, IntMod(3))

 poly(3 x - 2 y + 2, [x, y], IntMod(7)),

    poly(y - 1, [x, y], IntMod(3))
```

Example 9. Here we create a polynomial over the coefficient ring `Dom::Float`:

```
>> poly(3*x - y, Dom::Float)

            poly(- 1.0 y + 3.0 x, [y, x], Dom::Float)
```

The identifier `y` cannot turn up in coefficients from this ring, because it cannot be converted to a floating point number:

```
>> poly(3*x - y, [x], Dom::Float)

                  FAIL
```

Background:

- A domain must contain certain entries if it is to be used as a coefficient domain:

  - The entry "zero" must provide the neutral element with respect to addition.
  - The entry "one" must provide the neutral element with respect to multiplication.
  - The method "_plus" must add domain elements.
  - The method "_negate" must return the inverse with respect to addition.
  - The method "_mult" must multiply domain elements.
  - The method "_power" must compute integer powers of a domain element. It is called with the domain element as the first argument and an integer as the second argument.

- Further, the following methods should be defined. They are called by functions such as gcd, diff, divide, norm etc.:

  - The method "gcd" must return the greatest common divisor of domain elements.
  - The method "diff" must differentiate a domain element with respect to a variable.
  - The method "_divide" must divide two domain elements. It must return FAIL if division is not possible.
  - The method "norm" must compute the norm of a domain element and return it as a number.
  - The method "convert" must convert an expression to a domain element. It must return FAIL if this is not possible.

    This method is called to convert the coefficients of polynomial expressions to coefficients of the specified domain. If this method does not exist then only domain elements can be used to specify the coefficients.
  - The method "expr" must convert a domain element to an expression.

    The system function expr calls this method to convert a polynomial over the coefficient domain to a polynomial expression. If this method does not exist, domain elements are simply inserted into the expression.

- When converting a polynomial over a certain coefficient domain into a polynomial over the same domain, but a different set of indeterminates, the conversion can be made much more efficient if the domain has the axiom `Ax::indetElements`. It is implicitly assumed that this axiom holds for the domain `IntMod(n)`, but not for `Expr`.

Changes:

- The axiom `Ax::indetElements` is taken into acount.

`poly2list` – convert a polynomial to a list of terms

`poly2list(p)` returns a term list containing the coefficients and exponent vectors of the polynomial `p`.

Call(s):

- `poly2list(p)`
- `poly2list(f <, vars>)`

Parameters:

`p`	—	a polynomial of type `DOM_POLY`
`f`	—	a polynomial expression
`vars`	—	a list of indeterminates of the polynomial: typically, identifiers or indexed identifiers

Return Value: a list containing the coefficients and exponent vectors of the polynomial. `FAIL` is returned if a given expression cannot be converted to a polynomial.

Related Functions: `coeff`, `coerce`, `degree`, `degreevec`, `lcoeff`, `poly`, `tcoeff`

Details:

- The returned term list is a list where each element represents a monomial of the polynomial with non-zero coefficient. The monomials are also represented as lists, each containing two elements: The first element is the coefficient and the second the exponent or exponent vector of the monomial. If the polynomial is univariate, exponents are returned, otherwise exponent vectors are returned. Exponent vectors have the same form as returned by the function `degreevec`. A zero polynomial results in an empty list.

- The elements of the term list are sorted lexicographically according to the exponent vectors. This is also the ordering used internally for the terms of polynomials.
- `poly2list(f, vars)` is equivalent to `poly2list(poly(f, vars))`: First, the polynomial expression `f` is converted to a polynomial in the variables `vars` over the expressions. Then that polynomial is converted to a term list. If the variables `vars` are not given, the free identifiers contained in `f` are used as variables. See `poly` about details on how the expression is converted to a polynomial. `FAIL` is returned if the expression cannot be converted to a polynomial.
- `poly2list` is a function of the system kernel.

Example 1. The following expressions define univariate polynomials. Thus the term lists contain exponents and not exponent vectors:

```
>> poly2list(2*x^100 + 3*x^10 + 4)

                  [[2, 100], [3, 10], [4, 0]]
```

```
>> poly2list(2*x*(x + 1)^2)

                   [[2, 3], [4, 2], [2, 1]]
```

Specification of a list of indeterminates allows to distinguish symbolic parameters from the indeterminates:

```
>> poly2list(a*x^2 + b*x + c, [x])

                   [[a, 2], [b, 1], [c, 0]]
```

Example 2. In this example the polynomial is bivariate, thus exponent vectors are returned:

```
>> poly2list((x*(y + 1))^2, [x, y])

             [[1, [2, 2]], [2, [2, 1]], [1, [2, 0]]]
```

Example 3. In this example a polynomial of domain type `DOM_POLY` is given. This form must be used if the polynomial has coefficients that does not consist of expressions:

```
>> poly2list(poly(-4*x + 5*y - 5, [x, y], IntMod(7)))

            [[3, [1, 0]], [-2, [0, 1]], [2, [0, 0]]]
```

`polygon` – generate a graphical polygon primitive

`polygon(p1, p2, ...)` defines a polygon with vertices `p1`, `p2` etc.

Call(s):

- `polygon(p1, p2, ...` <, *Closed* `= b1`> <, *Filled* `= b2`> <, *Color* `= [r, g, b]`>`)`

Parameters:

`p1, p2, ...`	—	graphical points created by the function `point`. A 2D polygon is created if all points are 2D points. 3D points create a 3D polygon.

Options:

Closed `= b1`	—	`b1` may be either `TRUE` or `FALSE`. If `TRUE`, the first point `p1` is internally appended to the points, thus creating a closed polygon. The default is *Closed* `= FALSE`.
Filled `= b2`	—	`b2` may be either `TRUE` or `FALSE`. If `FALSE`, the polygon is a curve consisting of line segments. If `TRUE`, the polygon is rendered as a filled area. The default is *Filled* `= FALSE`.
Color `= [r, g, b]`	—	sets an RGB color given by the amount of red, green and blue. The parameters `r`, `g`, `b` must be real numbers between 0 and 1.

Return Value: an object of domain type `DOM_POLYGON`.

Related Functions: `plot`, `plot::Polygon`, `plot2d`, `plot3d`, `plotfunc2d`, `plotfunc3d`, `point`, `RGB`

Details:

- Polygons generated by `polygon` represent graphical primitives that can be displayed via `plot2d` or `plot3d` using the list format `[Mode = List, [..primitives..]]`.

- The `plot` library provides the alternative primitive `plot::Polygon`. This object is more flexible than the kernel object generated by `polygon`. The first can be used with all functions of the `plot` library, whereas the latter can only be used in a call to `plot2d` or `plot3d`.

- `polygon` is a function of the system kernel.

Option <`Filled = b2`>**:**

- With *`Filled`* `= TRUE` a closed polygon is created, i.e., the first point `p1` is appended to the points. The plot functions render the polygon as a filled area.
- If `Closed = FALSE`, the edges of the polygon are rendered with the same color as the interior. If `Closed = TRUE`, the edges are rendered in the foreground color of the scene.
- Filled 3D polygons may not consist of more than three points (triangles). Use `plot::Polygon` to generate more complex filled 3D polygons.

Option <`Color = [r, g, b]`>**:**

- The color values `r`, `g`, `b` must be numerical expressions that can be converted to real floating point numbers from the interval $[0.0, 1.0]$. An error occurs if any of these values is not in this range. Symbolic expressions such as `PI - 2`, `exp(-sqrt(2))` etc. are accepted. Note, however, that expressions involving symbolic identifiers are not accepted!
- Point colors may be specified in the definition of the vertices via the function `point`. These colors are ignored.
- The domain `RGB` contains many predefined colors.

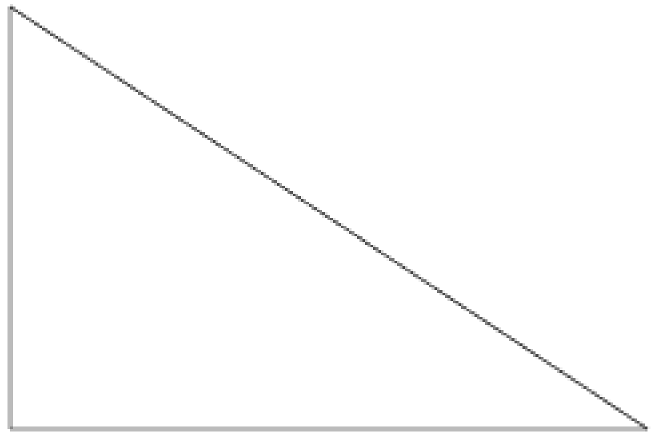
Operands: The first operands of a polygon are the vertices as specified in the generating call to `polygon`. The third but last operand is the list `[r, g, b]` defining the polygon color. This operand is `NIL`, if no color was specified. The second but last operand is the Boolean `b1` corresponding to *`Closed`* `= b1`. The last operand is the Boolean `b2` corresponding to *`Filled`* `= b2`.

Example 1. We define the vertices of a 2D triangle:

```
>> p1 := point(0, 0): p2 := point(0, 1): p3 := point(1, 0):
```

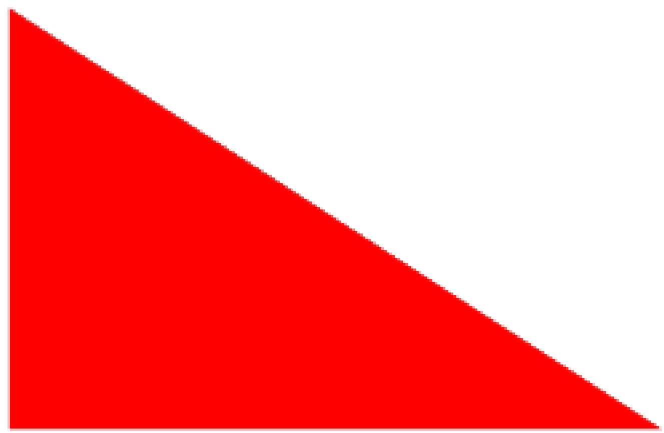
We use `plot2d` to render the edges of the triangle:

```
>> plot2d(Axes = None, [Mode = List,
          [polygon(p1, p2, p3, Closed = TRUE, Color = RGB::Black)]
          ])
```



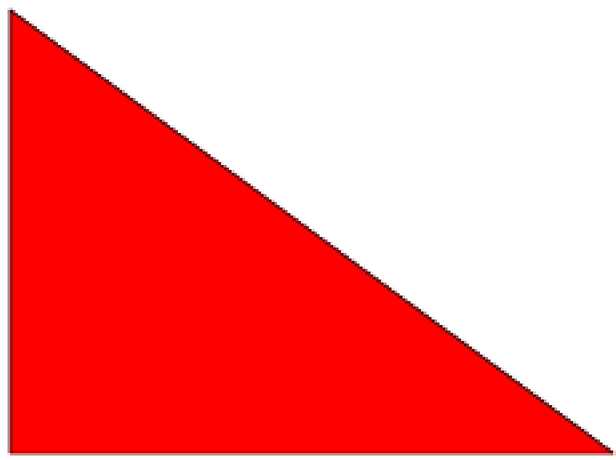
The following command renders the triangle area:

```
>> plot2d(Axes = None, [Mode = List,
          [polygon(p1, p2, p3, Filled = TRUE, Color = RGB::Red)]])
```



The following command renders the triangle area and the edges:

```
>> plot2d(Axes = None, [Mode = List,
          [polygon(p1, p2, p3, Closed = TRUE, Filled = TRUE,
           Color = RGB::Red)]])
```



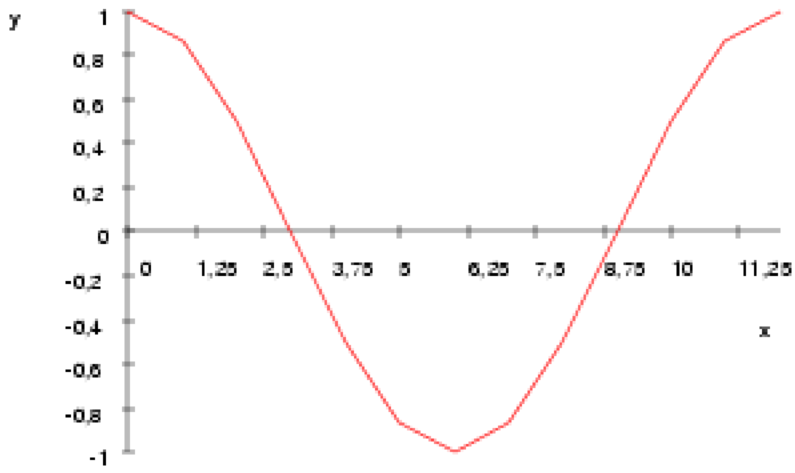
```
>> delete p1, p2, p3:
```

Example 2. We define 2D points on the graph of the cosine function:

```
>> for i from 0 to 12 do
     p[i] := point(i, cos(i*PI/6)):
   end_for:
```

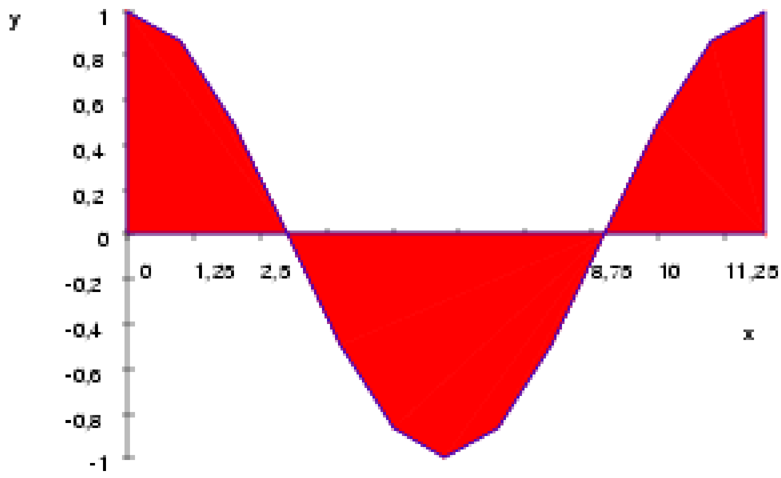
These points are used to build a polygon:

```
>> plot2d(Scaling = UnConstrained,
          [Mode = List, [polygon(p[i] $ i = 0..12)]])
```



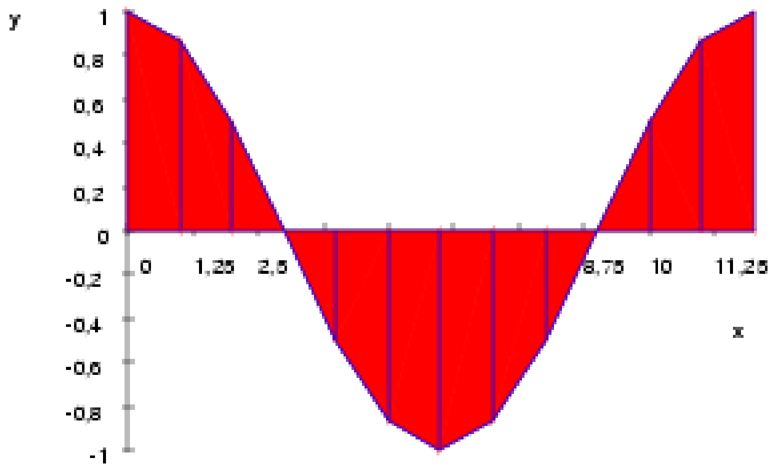
The following command plots the area between the graph of the cosine function and the x-axis:

```
>> plot2d(Scaling = UnConstrained, [Mode = List,
           [polygon(point(0, 0), p[i] $ i = 0..12, point(12, 0),
                    Closed = TRUE, Filled = TRUE)]])
```



The following command plots splits the area between the graph of the cosine function and the x-axis into trapezoids. The trapezoids are plotted as a list of filled polygons:

```
>> plot2d(Scaling = UnConstrained, [Mode = List,
          [polygon(point(i, 0), p[i], p[i+1], point(i + 1, 0),
                   Closed = TRUE, Filled = TRUE) $ i = 0..11]])
```



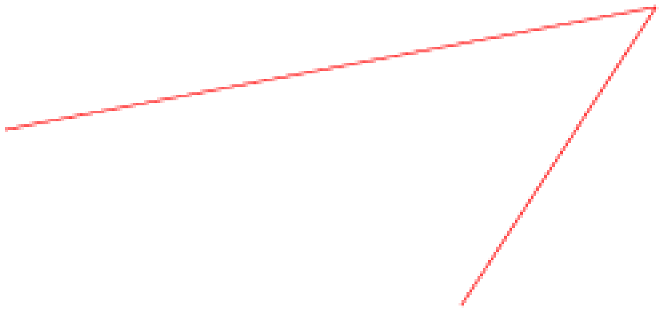

```
>> delete p:
```

Example 3. We define the vertices of a 3D triangle:

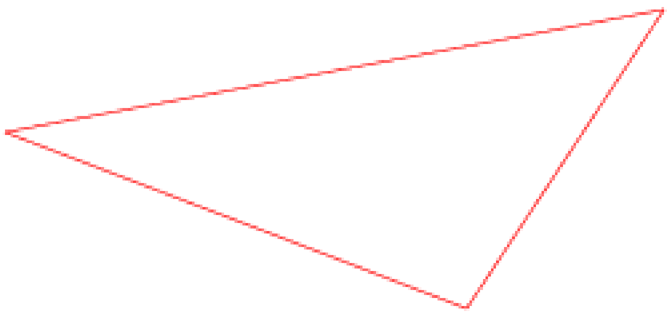
```
>> a := point(0, 0, 1): b := point(1, 1, 1): c := point(1, 0, 1):
```

We render the triangle in various modes:

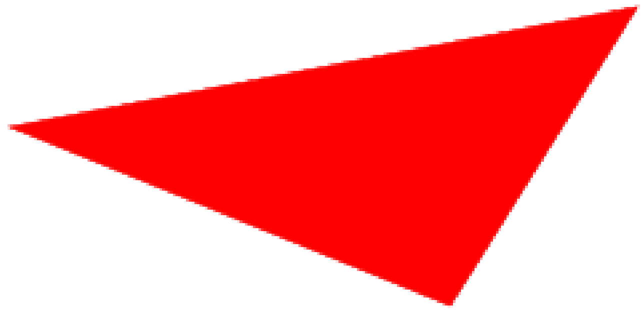
```
>> plot3d(Axes = None, [Mode = List, [polygon(a, b, c)] ])
```



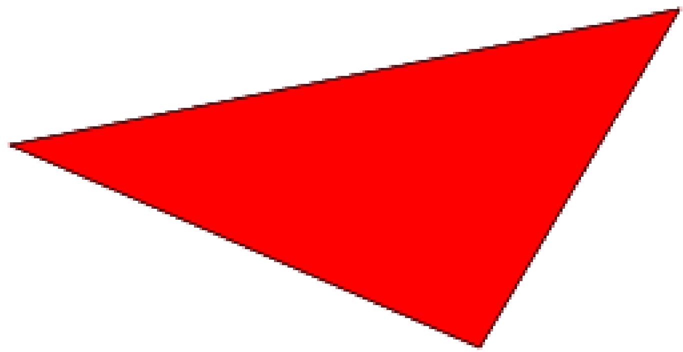
```
>> plot3d(Axes = None,
          [Mode = List, [polygon(a, b, c, Closed = TRUE)]])
```



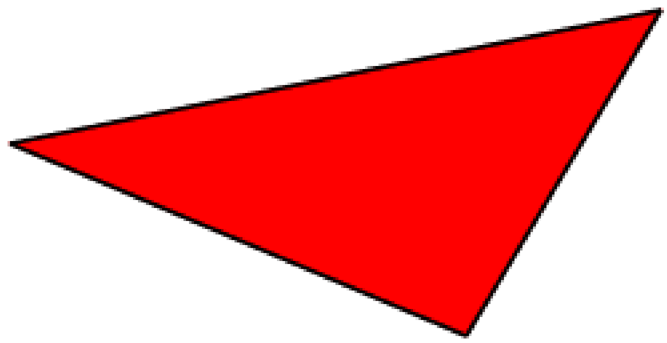
```
>> plot3d(Axes = None,
          [Mode = List, [polygon(a, b, c, Filled = TRUE)]])
```



```
>> plot3d(Axes = None, [Mode = List,
          [polygon(a, b, c, Closed = TRUE, Filled = TRUE)]])
```



```
>> plot3d(Axes = None, LineWidth = 30, [Mode = List,
          [polygon(a, b, c, Closed = TRUE, Filled = TRUE)]])
```



```
>> delete a, b, c:
```

---

polylog – the polylogarithm function

polylog(n,x) represents the polylogarithm function $Li_n(x)$ of index n at the point x.

Call(s):

- polylog(n, x)

Parameters:

n — an arithmetical expression representing an integer
x — an arithmetical expression

Return Value: an arithmetical expression.

Overloadable by: x

Side Effects: When called with a floating point argument `x`, the function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Related Functions: `dilog`, `ln`

Details:

- For a complex number x of modulus $|x| < 1$, the polylogarithm function of index n is defined as

$$Li_n(x) = \sum_{k=1}^{\infty} \frac{x^k}{k^n} \ .$$

  This function is extended to the whole complex plane by analytic continuation.

- If `n` is an integer and `x` a floating point number, then a floating point result is computed.

- If `n` is an integer ≤ 1, then an explicit expression is returned for any input parameter `x`. If `n` is an integer > 1 or if `n` is a symbolic expression, then an unevaluated call of `polylog` is returned, unless `x` is a floating point number. If `n` is a numerical value, but not an integer, then an error occurs.

- Some special values for $n = 2$ are implemented (cf. `dilog`). The values $Li_n(0) = 0$ and $Li_n(1) = \texttt{zeta}(n)$ are implemented for any n. Furthermore, $Li_n(-1) = (2^{1-n} - 1)\ \texttt{zeta}(n)$ for any $n \neq 1$.

- $Li_n(x)$ has a singularity at the point $x = 1$ for indices $n \leq 1$. For indices $n \geq 1$, the point $x = 1$ is a branch point. The branch cut is the real interval $[1, \infty)$. A jump occurs when crossing this cut. Cf. example 2.

- Mathematically, `polylog(2,x)` coincides with `dilog(1-x)`.

Example 1. Explicit results are returned for integer indices $n \leq 1$:

```
>> polylog(-5, x), polylog(-1, x), polylog(0, x), polylog(1, x)

            2        3        4    5
   x + 26 x  + 66 x  + 26 x  + x        x        x
   -----------------------------, --------, -----, -ln(1 - x)
                      6                   2  1 - x
              (1 - x)             (1 - x)
```

An unevaluated call is returned if the index is an integer $n > 1$ or a symbolic expression:

```
>> polylog(2, x), polylog(n^2 + 1, 2), polylog(n + 1, 2.0)

                                  2
     polylog(2, x), polylog(n  + 1, 2), polylog(n + 1, 2.0)
```

Floating point values are computed for integer indices n and floating point arguments x:

```
>> polylog(-5, -1.2), polylog(10, 100.0 + 3.2*I)

           -0.2326930882, 104.9131863 + 11.44600047 I
```

An error occurs if n is a numerical value, but not an integer:

```
>> polylog(5/2, x)

 Error: first argument must be an integer [polylog]
```

Some special symbolic values are implemented:

```
>> polylog(4, 1), polylog(5, -1), polylog(2, I)

                  4                               2
                PI     15 zeta(5)               PI
                ---, - ----------, I CATALAN - ---
                 90        16                    48

>> assume(n <> 1): polylog(n, -1)

                                         1 - n
                      - zeta(n) (1 - 2      )

>> unassume(n): polylog(n, -1)

                          polylog(n, -1)
```

Example 2. For indices $n \geq 1$, the real interval $[1, \infty)$ is a branch cut. The values returned by `polylog` jump when crossing this cut:

```
>> polylog(3, 1.2 + I/10^1000) - polylog(3, 1.2 - I/10^1000)

                          0.1044301529 I
```

Example 3. The functions `diff`, `float`, `limit`, and `series` handle expressions involving `polylog`:

```
>> diff(polylog(n, x), x), float(polylog(4, 3 + I))

         polylog(n - 1, x)
         -----------------, 3.177636803 + 1.859135861 I
                 x

>> series(polylog(4, sin(x)), x = 0)

             2       3       4          5       6
            x     25 x    13 x    1523 x    49 x        7
        x + -- - ----- - ----- + ------- + ----- + O(x )
            16    162     768     405000   51840
```

Background:

- The polylogarithms are characterized by $\frac{d}{dx}\, Li_n(x) = \frac{1}{x}\, Li_{n-1}(x)$ in conjunction with $Li_n(0) = 0$ and $Li_1(x) = -\ln(1-x)$. $Li_n(x)$ is a rational function in x for $n \leq 0$.
- Li_n has a branch cut along the real interval $[1, \infty)$ for indices $n \geq 1$. The value at a point x on the cut coincides with the limit "from below":

$$Li_n(x) = \lim_{\epsilon \to 0_+} Li_n(x - \epsilon\, i) = \lim_{\epsilon \to 0_+} Li_n(x + \epsilon\, i) - \frac{2\,\pi\, i}{(n-1)!}\ \ln(x)^{n-1}\ .$$

- Reference: L. Lewin, "Polylogarithms and Related Functions", North Holland (1981). L. Lewin (ed.), "Structural Properties of Polylogarithms", Mathematical Surveys and Monographs Vol. 37, American Mathematical Society, Providence (1991).

Changes:

- The `"series"` attribute was extended. Now, asymptotic expansions are available.

`powermod` – compute a modular power of a number or a polynomial

`powermod(b, e, m)` computes $b^e \bmod m$.

Call(s):

- `powermod(b, e, m)`

Parameters:

- `b` — the base: a number, or a polynomial of type `DOM_POLY`, or a polynomial expression
- `e` — the power: a nonnegative integer
- `m` — the modulus: a number, or a polynomial of type `DOM_POLY`, or a polynomial expression

Return Value: Depending on the type of `b`, the return value is a number, a polynomial or a polynomial expression. `FAIL` is returned if an expression cannot be converted to a polynomial.

Overloadable by: `b`

Related Functions: `_mod`, `divide`, `modp`, `mods`, `poly`

Details:

- If `b` and `m` are numbers, the modular power $b^e \bmod m$ can also be computed by the direct call `b^e mod m`. However, `powermod(b, e, m)` avoids the overhead of computing the intermediate result b^e and computes the modular power much more efficiently.
- If `b` is a rational number, then the modular inverse of the denominator is calculated and multiplied with the numerator.
- If the modulus `m` is an integer, then the base `b` must either be a number, a polynomial expression or a polynomial that is convertible to an `IntMod(m)`-polynomial.
- If the modulus `m` is a polynomial expression, then the base `b` must either be a number, a polynomial expression or a polynomial over the coefficient ring of MuPAD expressions.
- If the modulus `m` is a polynomial of domain type `DOM_POLY`, then the base `b` must either be a number, or a polynomial of the same type as `m` or a polynomial expression that can be converted to a polynomial of the same type as `m`.
- Note that the system function `_mod` in charge of modular arithmetic may be changed by the user; see the help page of `_mod`. The function `powermod` calls `_mod` and reacts accordingly. Cf. example 5.
- Internally, polynomials are divided by the function `divide`.

Example 1. We compute $3^{123456} \bmod 7$:

```
>> powermod(3, 123456, 7)

                                   1
```

If the base is a rational number, the modular inverse of the denominator is computed and multiplied with the numerator:

```
>> powermod(3/5, 1234567, 7)

                                   2
```

Example 2. The coefficients of the following polynomial expression are computed modulo 7:

```
>> powermod(x^2 + 7*x - 3, 10, 7)

            2     4      6    14    16      18    20
         3 x  - x  - 3 x  + x   - x   - 2 x   + x   - 3
```

Example 3. The power of the following polynomial expression is reduced modulo the polynomial $x^2 + 1$:

```
>> powermod(x^2 + 7*x - 3, 10, x^2 + 1)

                     1029668584 x - 534842913
```

Example 4. The type of the return value coincides with the type of the base: a polynomial is returned if the base is a polynomial:

```
>> powermod(poly(x^2 + 7*x - 3), 2, x^2 + 1),
   powermod(poly(x^2 + 7*x - 3), 2, poly(x^2 + 1))

         poly(- 56 x - 33, [x]), poly(- 56 x - 33, [x])
```

If the base is a polynomial expression, `powermod` returns a polynomial expression:

```
>> powermod(x^2 + 7*x - 3, 2, x^2 + 1),
   powermod(x^2 + 7*x - 3, 2, poly(x^2 + 1))

                     - 56 x - 33, - 56 x - 33
```

Example 5. The following re-definition of `_mod` switches to a symmetric representation of modular numbers:

```
>> alias(R = Dom::IntegerMod(17)):
   _mod := mods: powermod(poly(2*x^2, R), 3, poly(3*x + 1, R))

                      poly(-4, [x], R)
```

The following command restores the default representation:

```
>> _mod := modp: powermod(poly(2*x^2, R), 3, poly(3*x + 1, R))

                      poly(13, [x], R)

>> unalias(R):
```

`print` – print objects to the screen

`print(object)` displays `object` on the screen.

Call(s):

`print(<Unquoted,> <NoNL,> <KeepOrder,> object1, object2, ...)`

Parameters:

`object1, object2, ...` — any MuPAD objects

Options:

Unquoted	—	Display character strings without quotation marks and with expanded control characters `'\n'`, `'\t'`, and `'\\'`.
NoNL	—	Like *Unquoted*, but no newline is put at the end. `PRETTYPRINT` is implicitly set to `FALSE`.
KeepOrder	—	Display operands of sums (of type `"_plus"`) always in the internal order.

Return Value: `print` returns the void object `null()` of type `DOM_NULL`.

Overloadable by: `object1, object2, ...`

Side Effects: `print` is sensitive to the environment variables `DIGITS`, `PRETTYPRINT`, and `TEXTWIDTH`, and to the output preferences `Pref::floatFormat`, `Pref::keepOrder`, `Pref::matrixSeparator`, `Pref::timesDot`, and `Pref::trailingZeroes`.

Related Functions: `DIGITS`, `DOM_FUNC_ENV`, `expose`, `expr2text`, `finput`, `fprint`, `fread`, `funcenv`, `input`, `Pref::floatFormat`, `Pref::keepOrder`, `Pref::matrixSeparator`, `Pref::timesDot`, `Pref::trailingZeroes`, `PRETTYPRINT`, `protocol`, `read`, `TEXTWIDTH`, `userinfo`, `write`

Details:

- At interactive level, the result of a MuPAD command entered at the command prompt is usually displayed on the screen automatically. `print` serves to generate additional output from within loops or procedures.

- Apart from some exceptions mentioned below, the output generated by `print` is identical to the usual output of MuPAD results at interactive level.

- `print` evaluates its arguments sequentially from left to right (cf. example 2) and displays the results on the screen. The individual outputs are separated by commas. A new line is started at the end of the output if this is not suppressed by the option *NoNL*.

- The output width for `print` is limited by the environment variable `TEXTWIDTH`. Cf. example 3.

- The style of the output is determined by the value of the environment variable `PRETTYPRINT`. Cf. example 4. `print` will always produce ASCII output, even under Windows when typesetting is enabled for the result outputs.

- `print` descends recursively into the operands of an object. For each subobject `s`, `print` first determines its domain type `T`. If the domain `T` has a `"print"` slot, then `print` issues the call `T::print(s)` to the slot routine. In contrast to the overloading mechanism for most other MuPAD functions, `print` processes the result of this call recursively, and the result of the recursive process is printed at the position of `s` (cf. example 5).

  NOTE The result returned by the `"print"` method must not contain the domain element `s` itself as a subobject, since this leads to infinite recursion (cf. example 6). The same remark also applies to the output procedures of function environments (see below).

  If `T` is a built-in kernel domain without a `"print"` slot, then the output of `s` is handled by `print` itself.

  If `T` is a library domain without a `"print"` slot and the internal operands of `s` are `op1, op2, ...`, then `s` is printed as `new(T, op1, op2, ...)`. (See example 5.)

- Even the output of elements of a kernel domain can be changed by defining a `"print"` method. Cf. example 7.

- `"print"` methods may return strings or expressions. Strings are always printed unquoted. Expressions are printed in normal mode. If they contain strings, they will be printed with quotation marks. Cf. example 8.
- The output of an `expression` is determined by the 0th operand of the expression. If the 0th operand is a `function environment`, then its second operand handles the output of the expression (cf. examples 9 and 10). Otherwise, the expression is printed in functional notation.
- In contrast to the usual output of MuPAD objects at interactive level, `print` does not perform resubstitution of aliases (see `Pref::alias` for details). Moreover, the routines defined via `Pref::output` and `Pref::postOutput` are not called by `print`. Cf. example 15.
- The output of `floating point numbers` depends on the environment variable `DIGITS` and the settings of `Pref::floatFormat` (exponential or floating point representation) and `Pref::trailingZeroes` (printing of trailing zeroes). Cf. example 19.
- For an overview of all file related MuPAD functions, also try `?fileIO`.
- `print` is a function of the system kernel.

Option <`Unquoted`>**:**

- With this option, character strings are displayed without quotation marks. Moreover, the control characters `'\n'`, `'\t'`, and `'\\'` in strings are expanded into a new line, a tabulator skip, and a single backslash `'\'`, respectively. Cf. example 11.
- The control character `'\t'` is expanded with tab-size 8. The following character is placed in the next column `i` with `i mod 8 = 0`.

Option <`NoNL`>**:**

- This option has the same functionality as *`Unquoted`*. In addition, the new line at the end of the output is suppressed. Cf. example 13.
- Moreover, this option implicitly sets `PRETTYPRINT` to `FALSE`.

Option <`KeepOrder`>**:**

- This option determines the order of terms in sums. Normally, the system sorts the terms of a `sum` such that a positive term is in the first position of the output. If *`KeepOrder`* is given, no such re-ordering takes place and sums are printed in the internal order. Cf. example 14.

⊞ This behavior can also be controlled via `Pref::keepOrder`. More precisely, the call `print(`*KeepOrder*`, ...)` generates the same output as the following command:

```
Pref::keepOrder(Always): print(...): Pref::keepOrder(%2):
```

Example 1. This example shows a simple call of `print` with strings as arguments. They are printed with quotation marks:

```
>> print("Hello", "You"." !"):

                         "Hello", "You !"
```

Example 2. Like most other functions, `print` evaluates its arguments. In the following call, `x` evaluates to `0` and `cos(0)` evaluates to `1`:

```
>> a := 0: print(cos(a)^2):

                                  1
```

Use `hold` if you want to print the expression `cos(a)^2` literally:

```
>> print(hold(cos(a)^2)):

                                     2
                              cos(a)

>> delete a:
```

Example 3. `print` is sensitive to the current value of `TEXTWIDTH`:

```
>> print(expand((a + b)^4)):
   old := TEXTWIDTH: TEXTWIDTH := 30:
   print(expand((a + b)^4)):
   TEXTWIDTH := old:

               4    4        3      3        2  2
              a  + b  + 4 a b  + 4 a  b + 6 a  b

  4    4        3      3
 a  + b  + 4 a b  + 4 a  b +

      2  2
   6 a  b

>> delete old:
```

Example 4. `print` is sensitive to the current value of `PRETTYPRINT`:

```
>> print(a/b):
   old := PRETTYPRINT: PRETTYPRINT := FALSE:
   print(a/b):
   PRETTYPRINT := old:

                                a
                                -
                                b
 a/b

>> delete old:
```

Example 5. We demonstrate how to achieve formatted output for elements of a user-defined domain. Suppose that we want to write a new domain `Complex` for complex numbers. Each element of this domain has two operands: the real part `r` and the imaginary part `s`:

```
>> Complex := newDomain("Complex"): z := new(Complex, 1, 3):
   z + 1;
   print(z + 1):

                    (new(Complex, 1, 3)) + 1

                    (new(Complex, 1, 3)) + 1
```

Now we want a nicer output for elements of this domain, namely in the form `r+s*I`, where `I` denotes the imaginary unit. We implement the slot routine `Complex::print` to handle this. This slot routine will be called by MuPAD with an element of the domain `Complex` as argument whenever such an element is to be printed on the screen:

```
>> Complex::print := (z -> extop(z, 1) + extop(z, 2)*I):
   z + 1;
   print(z + 1):

                          (1 + 3 I) + 1

                          (1 + 3 I) + 1

>> delete Complex, z:
```

Example 6. The result of a "print" method must not contain the argument as a subobject; otherwise this leads to infinite recursion. In the following example, the slot routine T::print would be called infinitly often. MuPAD tries to trap such infinite recursions and prints ‘????‘ instead:

```
>> T := newDomain(T): T::print := id:
   new(T, 1);
   print(new(T, 1)):

                                  ‘????‘

                                  ‘????‘

>> delete T:
```

Example 7. Even "print" methods for kernel domains are possible. This example shows how to redefine the output of polynomials by printing only the polynomial expression:

```
>> poly(x + 1);
   print(poly(x + 1)):

                          poly(x + 1, [x])

                          poly(x + 1, [x])

>> unprotect(DOM_POLY): DOM_POLY::print := p -> op(p, 1):
   poly(x + 1);
   print(poly(x + 1)):
   delete DOM_POLY::print:  protect(DOM_POLY):

                                   x + 1

                                   x + 1
```

Example 8. If a "print" method returns a string, it will be printed unquoted:

```
>> Example := newDomain("Example"):  e := new(Example, 1):
   Example::print := x -> "elementOfExample":
   print(e):

                          elementOfExample
```

If a "print"-method returns an expression, it will be printed in normal mode. If the expression contains strings, they will be printed in the usual way with quotation marks:

```
>> Example::print := x -> ["elementOfExample", extop(x)]:
   print(e):

                  ["elementOfExample", 1]

>> delete Example, e:
```

Example 9. Suppose that you have defined a function `f` that may return itself symbolically, and you want such symbolic expressions of the form `f(x,...)` to be printed in a special way. To this end, embed your procedure `f` in a function environment and supply an output procedure as second argument to the corresponding `funcenv` call. Whenever an expression of the form `f(x,...)` is to be printed, the output procedure will be called with the arguments `x,...` of the expression:

```
>> f := funcenv(f,
            proc(x) begin
               if nops(x) = 2 then
                 "f does strange things with its arguments ".
                 expr2text(op(x, 1))." and ".expr2text(op(x,2))
               else
                 FAIL
               end
            end):

>> delete a, b:
   f(a, b)/2;
   f(a, b, c)/2

        f does strange things with its arguments a and b
        ------------------------------------------------
                                2

                           f(a, b, c)
                           ----------
                                2

>> delete f:
```

Example 10. For all prefedined `function environments`, the second operand is a built-in output function, of type `DOM_EXEC`. In particular, this is the case for operators such as `+`, `*`, `^` etc. In the following example, we change the output symbol for the power operator `^`, which is stored in the third operand of the built-in output function of the function environment `_power`, to a double asterisk:

```
>> unprotect(_power):
   _power := subsop(_power, [2, 3] = "**"):
   a^b/2;
   print(a^b/2):
   _power := subsop(_power, [2, 3] = "^"):
   protect(_power):
```

```
                              a**b
                              ----
                                2

                              a**b
                              ----
                                2
```

Example 11. With the option *Unquoted*, quotation marks are omitted:

```
>> print(Unquoted, "Hello", "You"." !"):
```

```
                          Hello, You !
```

With *Unquoted* the special characters `'\t'` and `'\n'` are expanded:

```
>> print(Unquoted, "As you can see\n".
                   "'\\n' is the newline character\n".
                   "\tand '\\t' a tabulator"):
```

```
                  As you can see
                  '\n' is the newline character
                          and '\t' a tabulator
```

Example 12. It is useful to construct output strings using `expr2text` and the concatenation operator `.`:

```
>> d := 5:  print(Unquoted, "d plus 3 = ".expr2text(d + 3)):
```

```
                          d plus 3 = 8
```

```
>> delete d:
```

Example 13. With the option *NoNL*, no new line is put at the end of the output and `PRETTYPRINT` is implicitly set to `FALSE`. Apart from that, the behavior is the same as with the option *Unquoted*:

```
>> print(NoNL, "Hello"):  print(NoNL, ",  You"." !\n"):
   print(NoNL, "As you can see PRETTYPRINT is FALSE: "):
   print(NoNL, x^2-1):  print(NoNL, "\n"):

 Hello,  You !
 As you can see PRETTYPRINT is FALSE: x^2 - 1
```

Example 14. If the option *KeepOrder* is given, sums are printed in their internal order:

```
>> print(b - a):  print(KeepOrder, b - a):

                              b - a

                             - a + b
```

Example 15. Alias resubstitution (see `Pref::alias`) takes place for normal result outputs in an interactive session, but not for outputs generated by `print`:

```
>> delete a, b: alias(a = b):
   a; print(a):
   unalias(a):

                                a

                                b
```

In contrast to the usual result output, `print` does not react to `Pref::output`:

```
>> old := Pref::output(generate::TeX):
   sin(a)^b; print(sin(a)^b):
   Pref::output(old):

                    "\\sin\\left(a\\right)^b"

                                   b
                             sin(a)
```

The same is true for `Pref::postOutput`:

```
>> old := Pref::postOutput("postOutput was called"):
   a*b; print(a*b):
   Pref::postOutput(old):
```

```
                              a b
 postOutput was called

                              a b

>> delete old:
```

Example 16. The output of summands of a sum depends on the form of these summands. If the summand is a `_mult` expression, only the first and last operand of the product are taken into account for determining the sign of that term in the output. If one of them is a negative number then the "+"-symbol in the sum is replaced by a "-"-symbol:

```
>> print(hold(a + b*c*(-2)),
         hold(a + b*(-2)*c),
         hold(a + (-2)*b*c)):

               a - 2 b c, a + b (-2) c, a - 2 b c
```

This has to be taken into account when writing `"print"`-methods for polynomial domains.

Example 17. Usually, MuPAD does not print a multiplication symbol for products and just concatenates the factors with spaces in between. You can explicitly request that a multiplication symbol be printed via `Pref::timesDot`:

```
>> a*b*c;
   print(a*b*c):

                             a b c

                             a b c

>> old := Pref::timesDot(" * "):
   a*b*c;
   print(a*b*c):
   Pref::timesDot(old):

                           a * b * c

                           a * b * c

>> delete old:
```

Example 18. The column separator in the output of matrices or two-dimensional arrays can be changed via `Pref::matrixSeparator`:

```
>> a := array(1..2, 1..2, [[11, 12], [22, 23]]):
   a; print(a):
```

```
                    +-        -+
                    |  11, 12  |
                    |          |
                    |  22, 23  |
                    +-        -+

                    +-        -+
                    |  11, 12  |
                    |          |
                    |  22, 23  |
                    +-        -+
```

```
>> old := Pref::matrixSeparator("  "):
   a; print(a):
   Pref::matrixSeparator(old): delete a:
```

```
                    +-        -+
                    |  11  12  |
                    |          |
                    |  22  23  |
                    +-        -+

                    +-        -+
                    |  11  12  |
                    |          |
                    |  22  23  |
                    +-        -+
```

If the output width of a matrix would exceed `TEXTWIDTH`, then it is printed in a textual form:

```
>> print(array(1..4, 1..4, (2, 2) = 2)):
   print(array(1..10, 1..10, (5, 5) = 55)):
```

```
         +-                                      -+
         |  ?[1, 1], ?[1, 2], ?[1, 3], ?[1, 4]  |
         |                                        |
         |  ?[2, 1],    2,    ?[2, 3], ?[2, 4]  |
         |                                        |
         |  ?[3, 1], ?[3, 2], ?[3, 3], ?[3, 4]  |
         |                                        |
         |  ?[4, 1], ?[4, 2], ?[4, 3], ?[4, 4]  |
```

```
+-                                      -+

          array(1..10, 1..10,
            (5, 5) = 55
          )
```

```
>> delete old, a:
```

Example 19. Floating point numbers are usually printed in fixed-point notation. You can change this to floating-point form with mantissa and exponent via `Pref::floatFormat`:

```
>> print(0.000001, 1000.0): old := Pref::floatFormat("e"):
   print(0.000001, 1000.0): Pref::floatFormat(old):

                         0.000001, 1000.0

                           1.0e-6, 1.0e3
```

In the default output of floating point numbers, trailing zeroes are cut off. This behavior can be changed via `Pref::trailingZeroes`:

```
>> print(0.000001, 1000.0): old := Pref::trailingZeroes(TRUE):
   print(0.000001, 1000.0): Pref::trailingZeroes(old):

                         0.000001, 1000.0

                  0.000001000000000, 1000.000000
```

The number of digits of floating point numbers in output depends on the environment variable `DIGITS`:

```
>> print(float(PI)):
   DIGITS := 20:  print(float(PI)):
   DIGITS := 30:  print(float(PI)):

                            3.141592654

                       3.1415926535897932385

                  3.14159265358979323846264338328
```

```
>> delete old, DIGITS:
```

Example 20. The output order of `sets` differs from the internal order of sets, which is returned by `op`:

```
>> s := {a, b, c}:
   s;
   print(s):
   op(s)
```

```
                    {a, b, c}

                    {a, b, c}

                     c, b, a
```

The index operator `[]` can be used to access the elements of a set with respect to the output order:

```
>> s[1], s[2], s[3]
```

```
                     a, b, c
```

```
>> delete s:
```

Example 21. The output of a domain is determined by its `"Name"` slot if it exists, and otherwise by its *key*:

```
>> T := newDomain("T"):
   T;
   print(T):
```

```
                        T

                        T
```

```
>> T::Name := "domain T":
   T;
   print(T):
```

```
                    domain T

                    domain T
```

```
>> delete T:
```

Example 22. It is sometimes desirable to combine strings with "pretty" expressions in an output. This is not possible via `expr2text`. On the other hand, an output with commas as separators is usually regarded as ugly. The following dummy expression sequence may be used to achieve the desired result. It uses MuPAD's internal function for standard operator output `builtin(1100,...)`, with priority 20—the priority of `_exprseq`—and with an empty operator symbol "":

```
>> myexprseq := funcenv(myexprseq,
                        builtin(1100, 20, "", "myexprseq")):
   print(Unquoted,
         myexprseq("String and pretty expression ", a^b, ".")):

                                                 b
                String and pretty expression a .

>> delete myexprseq:
```

Background:

- The output order of `sets` differs from the internal order of sets, which can be obtained via `op`. For this reordering in the output, the kernel calls the method `DOM_SET::sort`, which takes the set as argument and returns a sorted list. The elements of the set are then printed in the order given by this list.

`proc` – define a procedure

`proc - end_proc` defines a procedure.

Call(s):

- `(x1, x2, ...)  -> body`
- ```
  proc(
       x1 <= default1> <: type1>,
       x2 <= default2> <: type2>, ...
      )<: returntype>
  <name pname;>
  <option option1, option2, ...;>
  <local local1, local2, ...;>
  <save global1, global2, ...;>
  begin
    body
  end_proc
  ```
- `_procdef(...)`
  ```

Parameters:

`x1, x2, ...`	—	the formal parameters of the procedure: identifiers
`default1, default2, ...`	—	default values for the parameters: arbitrary MuPAD objects
`type1, type2, ...`	—	admissible types for the parameters: type objects as accepted by the function `testtype`
`returntype`	—	admissible type for the return value: a type object as accepted by the function `testtype`
`pname`	—	the name of the procedure: an expression
`option1, option2, ...`	—	available options are: ***escape***, ***hold***, ***noDebug***, ***remember***
`local1, local2, ...`	—	the local variables: identifiers
`global1, global2, ...`	—	global variables: identifiers
`body`	—	the body of the procedure: an arbitrary sequence of statements

Return Value: a procedure of type `DOM_PROC`.

Related Functions: `args`, `context`, `debug`, `expose`, `hold`, `MAXDEPTH`, `newDomain`, `Pref::ignoreNoDebug`, `Pref::noProcRemTab`, `Pref::typeCheck`, `Pref::warnDeadProcEnv`, `return`, `save`, `testargs`, `Type`

Details:

- Procedures `f := proc(x1, x2, ...) ... end_proc` may be called like a system function in the form `f(x1, x2, ...)`. The return value of this call is the value of the last command executed in the procedure body (or the value returned by the body via the function `return`).

- The procedure declaration `(x1, x2, ...) -> body` is equivalent to `proc(x1, x2, ...) begin body end_proc`. It is useful for defining *simple* procedures that do not need local variables. E.g., `f := x -> x^2` defines the mathematical function $f: \ x \mapsto x^2$. If the procedure uses more than one parameter, use brackets as in `f := (x, y) -> x^2 + y^2`. Cf. example 1.

- A MuPAD procedure may have an arbitrary number of parameters. For each parameter, a default value may be specified. This value is used if no actual value is passed when the procedure is called. E.g.,

  `f := proc(x = 42) begin body end_proc`

  defines the default value of the parameter `x` to be `42`. The call `f()` is equivalent to `f(42)`. Cf. example 2.

- For each parameter, a type may be specified. This invokes an automatic type checking when the procedure is called. E.g.,

  `f := proc(x : DOM_INT) begin body end_proc`

  restricts the argument `x` to integer values. If the procedure is called with an argument of a wrong data type, the evaluation is aborted with an error message. Cf. example 3. Checking the input parameters should be a standard feature of every procedure. Also refer to `testargs`.

  Also an automatic type checking for the return value may be implemented specifying `returntype`. Cf. example 3.

- With the keyword `name`, a name may be defined for the procedure, e.g.,

  `f := proc(...) name myName; begin body end_proc`.

  There is a special variable `procname` associated with a procedure which stores its name. When the body returns a symbolic call `procname(args())`, the actual name is substituted. This is the name defined by the optional `name` entry. If no `name` entry is specified, the first identifier the procedure has been assigned to is used as the name, i.e., `f` in this case. Cf. example 4.

- With the keyword `option`, special features may be specified for a procedure:

  escape must be used if the procedure creates and returns a new procedure which accesses local values of the enclosing procedure. Cf. example 5. This option should only be used if necessary. Also refer to `Pref::warnDeadProcEnv`.

  hold prevents the procedure from evaluating the actual parameters it is called with. Cf. example 6.

  noDebug prevents the MuPAD source code debugger from entering this procedure. Also refer to `Pref::ignoreNoDebug`. Cf. example 7.

  remember instructs the procedure to store each computed result in a so-called remember table. When this procedure is called later with the same input parameters, the result is read from this table and needs not be computed again. This may speed up, e.g., recursive procedures drastically. Cf. example 8. However, the remember table may grow large and use a lot of memory. Also refer to `Pref::noProcRemTab`.

- With the keyword `local`, the local variables of the procedure are specified, e.g.,

  `f := proc(...) local x, y; begin body end_proc`.

  Cf. example 9.

  Local variables cannot be used as "symbolic variables" (identifiers). They must be assigned values before they can be used in computations.

Note that the names of global MuPAD variables such DIGITS, READPATH etc. should not be used as local variables. Also refer to the keyword save.

- With the keyword save, a local context for global MuPAD variables is created, e.g.,

  `f := proc(...) save DIGITS; begin DIGITS := newValue; ... end_proc.`

  This means that the values these variables have on entering the procedure are restored on exiting the procedure. This is true even if the procedure is exited because of an error. Cf. example 10.

- One can define procedures that accept a variable number of arguments. E.g., one may declare the procedure without any formal parameters. Inside the body, the actual parameters the procedure is called with may be accessed via the function args. Cf. example 11.

- Calling a procedure name f, say, usually does not print the source code of the body to the screen. Use expose(f) to see the body. Cf. example 12.

- The environment variable MAXDEPTH limits the "nesting depth" of recursive procedure calls. The default value is MAXDEPTH $= 500$. Cf. example 8.

- If a procedure is a domain slot, the special variable dom contains the name of the domain the slot belongs to. If the procedure is not a domain slot, the value of dom is NIL.

- Instead of end_proc, also the keyword end can be used.

- The imperative declaration proc - end_proc internally results in a call of the kernel function _procdef. There is no need to call _procdef directly.

- _procdef is a function of the system kernel.

Example 1. Simple procedures can be generated with the "arrow operator" ->:

```
>> f := x -> x^2 + 2*x + 1:
   f(x), f(y), f(a + b), f(1.5)

           2             2                          2
   2 x + x  + 1, 2 y + y  + 1, 2 a + 2 b + (a + b)  + 1, 6.25

>> f := n -> isprime(n) and isprime(n + 2):
   f(i) $ i = 11..18

      TRUE, FALSE, FALSE, FALSE, FALSE, FALSE, TRUE, FALSE
```

The following command maps an "anonymous" procedure to the elements of a list:

```
>> map([1, 2, 3, 4, 5, 6], x -> x^2)
```

```
                    [1, 4, 9, 16, 25, 36]
```

```
>> delete f:
```

Example 2. The declaration of default values is demonstrated. The following procedure uses the default values if the procedure call does not provide all parameter values:

```
>> f := proc(x, y = 1, z = 2) begin [x, y, z] end_proc:
   f(x, y, z), f(x, y), f(x)

                 [x, y, z], [x, y, 2], [x, 1, 2]
```

No default value was declared for the first argument. A warning is issued if this argument is missing:

```
>> f()

 Warning: Uninitialized variable 'x' used;
 during evaluation of 'f'

                           [NIL, 1, 2]

>> delete f:
```

Example 3. The automatic type checking of procedure arguments and return values is demonstrated. The following procedure accepts only positive integers as argument:

```
>> f := proc(n : Type::PosInt) begin n! end_proc:
```

An error is raised if an unsuitable parameter is passed:

```
>> f(-1)

 Error: Wrong type of 1. argument (type 'Type::PosInt' expected,
        got argument '-1');
 during evaluation of 'f'
```

In the following procedure, automatic type checking of the return value is invoked:

```
>> f := proc(n : Type::PosInt) : Type::Integer
   begin
     n/2
   end_proc:
```

An error is raised if the return value is not an integer:

```
>> f(3)

 Error: Wrong type of return value (type 'Type::Integer' expected,
        value is '3/2');
 during evaluation of 'f'

>> delete f:
```

Example 4. The `name` entry of procedures is demonstrated. A procedure returns a symbolic call to itself by using the variable `procname` that contains the current procedure name:

```
>> f := proc(x)
   begin
     if testtype(x,Type::Numeric)
       then return(float(1/x))
       else return(procname(args()))
     end_if
   end_proc:
   f(x), f(x + 1), f(3), f(2*I)

              f(x), f(x + 1), 0.3333333333, -0.5 I
```

Also error messages use this name:

```
>> f(0)

 Error: Division by zero;
 during evaluation of 'f'
```

If the procedure has a name entry, this entry is used:

```
>> f := proc(x)
   name myName;
   begin
     if testtype(x,Type::Numeric)
       then return(float(1/x))
       else return(procname(args()))
     end_if
   end_proc:
   f(x), f(x + 1), f(3), f(2*I)

         myName(x), myName(x + 1), 0.3333333333, -0.5 I

>> f(0)
```

```
 Error: Division by zero;
 during evaluation of 'myName'

>> delete f:
```

Example 5. The option *escape* is demonstrated. This option must be used if the procedure returns another procedure that references a formal parameter or a local variable of the generating procedure:

```
>> f := proc(n)
   begin
     proc(x) begin x^n end_proc
   end_proc:
```

Without the option *escape*, the formal parameter `n` of `f` leaves its scope: `g := f(3)` references `n` internally. When `g` is called, it cannot evaluate `n` to the value `3` that `n` had inside the scope of the function `f`:

```
>> g := f(3): g(x)

 Warning: Uninitialized variable 'unknown' used;
 during evaluation of 'g'
 Error: Illegal operand [_power];
 during evaluation of 'g'
```

`option` *escape* instructs the procedure `f` to deal with variables escaping the local scope. Now, the procedure `g := f(3)` references the value `3` rather than the formal parameter `n` of `f`, and `g` can be executed correctly:

```
>> f := proc(n)
   option escape;
   begin
     proc(x) begin x^n end_proc
   end_proc:
   g := f(3): g(x), g(y), g(10)
```

$$x^3, y^3, 1000$$

```
>> delete f, g:
```

Example 6. The option *hold* is demonstrated. With `hold`, the procedure sees the actual parameter in the form that was used in the procedure call. Without `hold`, the function only sees the value of the parameter:

```
>> f := proc(x) option hold; begin x end_proc:
   g := proc(x) begin x end_proc:
   x := PI/2:
   f(sin(x) + 2) = g(sin(x) + 2), f(1/2 + 1/3) = g(1/2 + 1/3)

                  sin(x) + 2 = 3, 1/2 + 1/3 = 5/6
```

Procedures using `option` *`hold`* can evaluate the arguments with the function `context`:

```
>> f := proc(x) option hold; begin x = context(x) end_proc:
   f(sin(x) + 2), f(1/2 + 1/3)

                  sin(x) + 2 = 3, 1/2 + 1/3 = 5/6

>> delete f, g, x:
```

Example 7. The option *`noDebug`* is demonstrated. The `debug` command starts the debugger which steps inside the procedure `f`. After entering the debugger command `c` (continue), the debugger continues the evaluation:

```
>> f := proc(x) begin x end_proc: debug(f(42))

 Activating debugger...

 #0 in f($1=42) at /tmp/debug0.556:4
 mdx> c
 Execution completed.

                              42
```

With the option *`noDebug`*, the debugger does not step into the procedure:

```
>> f := proc(x) option noDebug; begin x end_proc: debug(f(42))

 Execution completed.

                              42

>> delete f:
```

Example 8. The option *`remember`* is demonstrated. The `print` command inside the following procedure indicates if the procedure body is executed:

```
>> f:= proc(n : Type::PosInt)
   option remember;
   begin
      print("computing ".expr2text(n)."!");
      n!
   end_proc:
   f(5), f(10)
```

```
"computing 5!"

"computing 10!"

120, 3628800
```

When calling the procedure again, all values that were computed before are taken from the internal "remember table" without executing the procedure body again:

```
>> f(5)*f(10) + f(15)
```

```
"computing 15!"

1308109824000
```

`option` *remember* is used in the following procedure which computes the Fibonacci numbers $F(0) = 0, F(1) = 1, F(n) = F(n-1) + F(n-2)$ recursively:

```
>> f := proc(n : Type::NonNegInt)
   option remember;
   begin
      if n = 0 or n = 1 then return(n) end_if;
      f(n - 1) + f(n - 2)
   end_proc:
```

```
>> f(123)
```

```
22698374052006863956975682
```

Due to the recursive nature of `f`, the arguments are restricted by the maximal recursive depth (see `MAXDEPTH`):

```
>> f(1000)
```

```
Error: Recursive definition [See ?MAXDEPTH];
during evaluation of 'Type::testtype'
```

Without `option` *remember*, the recursion is rather slow:

```
>> f := proc(n : Type::NonNegInt)
   begin
      if n = 0 or n = 1 then return(n) end_if;
      f(n - 1) + f(n - 2)
   end_proc:
```

```
>> f(28)
```

317811

```
>> delete f:
```

Example 9. We demonstrate the use of local variables:

```
>> f := proc(a)
   local x, y;
   begin
     x := a^2;
     y := a^3;
     print("x, y" = (x, y));
     x + y
   end_proc:
```

The local variables `x` and `y` do not coincide with the global variables `x`, `y` outside the procedure. The call to `f` does not change the global values:

```
>> x := 0: y := 0: f(123), x, y
```

"x, y" = (15129, 1860867)

1875996, 0, 0

```
>> delete f, x, y:
```

Example 10. The `save` declaration is demonstrated. The following procedure changes the environment variable `DIGITS` internally. Because of `save DIGITS`, the original value of `DIGITS` is restored after return from the procedure:

```
>> myfloat := proc(x, digits)
   save DIGITS;
   begin
     DIGITS := digits;
     float(x);
   end_proc:
```

The current value of `DIGITS` is:

```
>> DIGITS
```

With the default setting DIGITS = 10, the following float conversion suffers from numerical cancellation. Due to the higher internal precision, `myfloat` produces a more accurate result:

```
>> x := 10^20*(PI - 21053343141/6701487259):
   float(x), myfloat(x, 20)

                      -32.0, 0.02616403997
```

The value of `DIGITS` was not changed by the call to `myfloat`:

```
>> DIGITS

                                   10
```

The following procedure needs a global identifier, because local variables cannot be used as integration variables in the `int` function. Internally, the global identifier `x` is deleted to make sure that `x` does not have a value:

```
>> f := proc(n)
   save x;
   begin
     delete x;
     int(x^n*exp(-x), x = 0..1)
   end_proc:

>> x := 3: f(1), f(2), f(3)

          1 - 2 exp(-1), 2 - 5 exp(-1), 6 - 16 exp(-1)
```

Because of `save x`, the previously assigned value of `x` is restored after the integration:

```
>> x

                                   3

>> delete myfloat, x, f:
```

Example 11. The following procedure accepts an arbitrary number of arguments. It accesses the actual parameters via `args`, puts them into a list, reverses the list via `revert`, and returns its arguments in reverse order:

```
>> f := proc()
   local arguments;
   begin
      arguments := [args()];
      op(revert(arguments))
   end_proc:
```

```
>> f(a, b, c)
```

```
                    c, b, a
```

```
>> f(1, 2, 3, 4, 5, 6, 7)
```

```
              7, 6, 5, 4, 3, 2, 1
```

```
>> delete f:
```

Example 12. Use `expose` to see the source code of a procedure:

```
>> f := proc(x = 0, n : DOM_INT)
   begin
     sourceCode;
   end_proc
```

```
              proc f(x, n) ... end
```

```
>> expose(f)
```

```
            proc(x = 0, n : DOM_INT)
              name f;
            begin
              sourceCode
            end_proc
```

```
>> delete f:
```

`product` – definite and indefinite products

`product(f, i)` computes the indefinite product of $f(i)$ with respect to i, i.e., a closed form g such that $g(i+1)/g(i) = f(i)$.

`product(f, i = a..b)` tries to find a closed form representation of the product $\prod_{i=a}^{b} f(i)$.

Call(s):

- `product(f, i)`
- `product(f, i = a..b)`

Parameters:

`f`	—	an arithmetical expression depending on `i`
`i`	—	the product index: an identifier
`a, b`	—	the boundaries: arithmetical expressions

Return Value: an arithmetical expression.

Related Functions: `_mult`, `*`, `sum`

Details:

- `product` serves for simplifying *symbolic* products. It should *not* be used for multiplying a finite number of terms: if `a` and `b` are integers of type `DOM_INT`, the call `_mult(f $ i = a..b)` is more efficient than `product(f, i = a..b)`.

- `product(f, i)` computes the indefinite product of `f` with respect to `i`. This is an expression `g` such that $f(i) = g(i+1)/g(i)$.

- `product(f, i = a..b)` computes the definite product with `i` running from `a` to `b`.

  If `b-a` is a nonnegative integer then the explicit product $f(a) \cdot f(a+1) \cdots f(b)$ is returned.

  If `b-a` is a negative integer, then the reciprocal of the result of `product(f, i = b+1..a-1)` is returned. If the latter is zero, then the system issues an error message. With this convention, the rule

  `product(f, i = a..b) * product(f, i = b+1..c) = product(f, i = a..c)`

  is satisfied for any `a`, `b`, and `c`.

- The system returns a symbolic `product` call if it cannot compute a closed form representation of the product.

Example 1. Each of the following two calls computes the product $1 \cdot 2 \cdot 3 \cdot 4 \cdot 5$:

```
>> product(i, i = 1..5) = _mult(i $ i = 1..5)

                       120 = 120
```

However, using `_mult` is usually more efficient when the boundaries are integers of type `DOM_INT`.

There is a closed form of this definite product from 1 to n:

```
>> product(i, i = 1..n)

                     gamma(n + 1)
```

Since the upper boundary is a symbolic identifier `n`, `_mult` cannot handle this product:

```
>> _mult(i $ i = 1..n)
```

```
 Error: Illegal argument [_seqgen]
```

The corresponding indefinite product is:

```
>> product(i, i);

                              gamma(i)
```

The indefinite and the definite product of $2i + 1$ are:

```
>> product(2*i + 1, i)

                            i
                           2  gamma(i + 1/2)

>> product(2*i + 1, i = 1..n)

                          n + 1
                         2      gamma(n + 3/2)
                         ---------------------
                                    1/2
                                  PI
```

The boundaries may be symbolic expressions or $\pm\infty$ as well:

```
>> product(2*i/(i + 2), i = a..b)

                                            b + 1
                   gamma(a + 2) gamma(b + 1) 2
                   --------------------------------
                                            a
                       gamma(a) gamma(b + 3) 2

>> product(i^2/(i^2 - 1), i = 2..infinity)

                                    2
```

The system cannot find closed forms of the following two products and returns symbolic `product` calls:

```
>> delete f: product(f(i), i)

                           product(f(i), i)

>> product((1 + 2^(-i)), i = 1..infinity)

                         /  1                        \
                  product| -- + 1, i = 1..infinity |
                         |  i                        |
                         \ 2                         /
```

`protect` – **protect an identifier**

`protect(x)` protects the identifier `x`.

Call(s):

- `protect(x <, protectionlevel>)`

Parameters:

`x` — an identifier

Options:

`protectionlevel` — one of the flags *ProtectLevelError*, *ProtectLevelWarning*, or *ProtectLevelNone*. The default value is *ProtectLevelWarning*.

Return Value: the previous protection level of `x`: either *ProtectLevelError* or *ProtectLevelWarning* or *ProtectLevelNone*.

Related Functions: `unprotect`

Details:

- `protect(x, ProtectLevelError)` sets full write-protection for the identifier. Any subsequent attempt to assign a value to the identifier will lead to an error.
- `protect(x, ProtectLevelWarning)` sets a "soft" protection. Any subsequent assignment to the identifier results in a warning message. However, the identifier will be assigned a value, anyway.

  `protect(x)` is equivalent to `protect(x, ProtectLevelWarning)`.
- `protect(x, ProtectLevelNone)` removes any protection from the identifier. This call is equivalent to `unprotect(x)`.
- Overwriting protected identifiers such as the names of MuPAD functions may damage your current session.

Example 1. The following call protects the identifier `important` with the protection level "*ProtectLevelWarning*":

```
>> protect(important, ProtectLevelWarning)
```

ProtectLevelNone

The identifier can still be overwritten:

```
>> important := 1
```

```
Warning: protected variable important overwritten
```

1

We protect the identifier with the level "*ProtectLevelError*":

```
>> protect(important, ProtectLevelError)
```

ProtectLevelWarning

Now, it is no longer possible to overwrite `important`:

```
>> important := 2
```

```
Error: Identifier 'important' is protected [_assign]
```

The identifier keeps its previous value:

```
>> important
```

1

In order to overwrite this value, we must unprotect `important`:

```
>> protect(important, ProtectLevelNone)
```

ProtectLevelError

```
>> important := 2
```

2

The identifier is protected again with the default level "*ProtectLevelWarning*"':

```
>> protect(important)
```

ProtectLevelNone

```
>> important := 1
```

```
Warning: protected variable important overwritten
```

1

```
>> unprotect(important): delete important:
```

Example 2. `protect` does not evaluate its first argument. Here the identifier `x` can still be overwritten, while its value – which is the identifier `y` – remains write protected:

```
>> protect(y, ProtectLevelError): x := y: protect(x): x := 1

 Warning: protected variable x overwritten

                                  1

>> y := 2

 Error: Identifier 'y' is protected [_assign]

>> unprotect(x): unprotect(y): delete x, y:
```

Background:

- `protect` does not evaluate its first argument. This way identifiers can be protected that have been assigned a value.
- Identifiers starting with a `#` are implicitly protected and cannot be assigned a value nor receive assumptions.

Changes:

- The options *None*, *Warning* and *Error* were renamed to *ProtectLevelNone*, *ProtectLevelWarning* and *ProtectLevelError*.

`protocol` – create a protocol of a MuPAD session

`protocol(filename)` starts a protocol of the current MuPAD session in the file with the name `filename`.

`protocol(n)` writes into the file associated with the file descriptor `n`.

`protocol()` stops the protocol.

Call(s):

- `protocol(filename <, InputOnly>)`
- `protocol(n <, InputOnly>)`
- `protocol()`

Parameters:

`filename`	—	the name of a file: a character string
`n`	—	a file descriptor provided by `fopen`: a positive integer

Options:

InputOnly — only input is protocolled

Return Value: the void object of type `DOM_NULL`.

Side Effects: The function is sensitive to the environment variable `WRITEPATH`. If this variable has a value, then the protocol file is created in the corresponding directory. Otherwise, the file is created in the "current working directory".

Related Functions: `fclose`, `fileIO`, `finput`, `fname`, `fopen`, `fprint`, `fread`, `ftextinput`, `pathname`, `print`, `read`, `READPATH`, `write`, `WRITEPATH`

Details:

- `protocol` writes a protocol of input commands and corresponding MuPAD output to a text file.
- The file may be specified directly by its name. This either creates a new file or overwrites an existing file. `protocol` opens and closes the file automatically.

  If `WRITEPATH` does not have a value, `protocol` interprets the file name as a pathname relative to the "working directory".

  Note that the meaning of "working directory" depends on the operating system. On Windows systems, the "working directory" is the folder where MuPAD is installed. On UNIX or Linux systems, it is the current working directory in which MuPAD was started.

  On the Macintosh, an empty file name may be given. In this case, a dialogue box is opened in which the user can choose a file. Further, on the interactive level, MacMuPAD warns the user, if an existing file is about to be overwritten.

  Also absolute path names are processed by `protocol`.
- Alternatively, the file may be specified by a file descriptor `n`. In this case, the file must have been opened via `fopen(Text, filename, Write)` or `fopen(Text, filename, Append)`. This returns the file descriptor as an integer `n`. Note that `fopen(filename)` opens the file in read-only mode. A subsequent `protocol` command to this file causes an error.

  The file is not closed automatically by `protocol()` and must be closed by a subsequent call to `fclose`.
- A call of `protocol` without arguments terminates a running protocol and closes the corresponding file. Closing the protocol file with `fclose` also terminates the protocol.
- If a new protocol is started while a protocol is running, then the old one is terminated and the corresponding file is closed.
- For an overview of all file related MuPAD functions, also try `?fileIO`.

Option <InputOnly>:

- The protocol file only contains the input lines. All output is omitted.

Example 1. We open a text file `test` in write mode with `fopen`:

```
>> n := fopen(Text, "test", Write):
```

A protocol is written into this file:

```
>> protocol(n):
   1 + 1, a/b;
   solve(x^2 = 2);
   protocol():
```

The file now has the following content:

```
 1 + 1, a/b;

                                                a
                                             2, -
                                                b
 solve(x^2 = 2)

                                   1/2           1/2
                             {[x = 2   ], [x = - 2   ]}
 protocol():
```

Example 2. The protocol file is opened directly by `protocol`. Only input is protocolled:

```
>> protocol("test", InputOnly):
   1 + 1; a/b;
   solve(x^2 = 2);
   protocol():
```

The file now has the following content:

```
 1 + 1; a/b;
 solve(x^2 = 2);
 protocol():
```

`psi` – the digamma/polygamma function

`psi(x)` represents the digamma function, i.e., the logarithmic derivative $\Psi(x) = \Gamma'(x)/\Gamma(x)$ of the `gamma` function.

`psi(x, n)` represents the n-th polygamma function, i.e., the n-th derivative $\Psi^{(n)}(x)$.

Call(s):

- `psi(x)`
- `psi(x, n)`

Parameters:

`x` — an arithmetical expression
`n` — a nonnegative integer

Return Value: an arithmetical expression.

Overloadable by: `x`

Side Effects: When called with a floating point value `x`, the function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Related Functions: `beta`, `binomial`, `fact`, `gamma`, `zeta`

Details:

- `psi(x, 0)` is equivalent to `psi(x)`.
- The digamma/polygamma function is defined for all complex arguments `x` apart from the singular points $0, -1, -2, \ldots$.
- If `x` is a floating point value, then a floating point value is returned.

  Simplifcations are implemented for rational numbers x with $|x| < 500$. In particular, if $x =$ `numer(x)`$/k$ with denominators $k = 1$, 2, 3, 4 or 6, the functional equation

  $$\Psi^{(n)}(x+1) = \Psi^{(n)}(x) + \frac{(-1)^n n!}{x^{n+1}},$$

  is used to obtain a result with an argument `x` from the interval $(0, 1]$.

  Some explicit formulas are implemented including `psi(1) = -EULER`,

`psi(1, n) = (-1)^(n + 1)*n!*zeta(n + 1)`, `n > 0`,

`psi(1/2) = -2*ln(2) - EULER`,

`psi(1/2, n) = (-1)^(n + 1)*n!*(2^(n + 1) - 1)*zeta(n + 1)`, `n > 0`.

The special values

`psi(infinity) = psi(infinity, 0) = infinity`

and

`psi(infinity, n) = 0` for `n > 0`

are implemented.

For all other arguments, a symbolic function call of `psi` is returned.

- The float attribute of the digamma function `psi(x)` is a kernel function, i.e., floating point evaluation is fast. The float attribute of the polygamma function `psi(x, n)` with `n > 0` is a library function. Note that `psi(float(x))` and `psi(float(x), n)` rather than `float(psi(x))` and `float(psi(x, n))` should be used for float evaluation because, for rational values of `x`, the computation of the symbolic result `psi(x)`, `psi(x, n)` may be costly. Further, the float evaluation of the symbolic result may be numerically unstable.

- The `expand` attribute uses the functional equation

$$\Psi^{(n)}(x+1) = \Psi^{(n)}(x) + \frac{(-1)^n n!}{x^{n+1}},$$

the nth derivative of the reflection formula

$$\Psi(-x) = \Psi(x) + \frac{1}{x} + \pi\cot(\pi x),$$

and the Gauß multiplication formula for $\Psi^{(n)}(kx)$ when k is a positive integer, to rewrite `psi(x, n)`. For numerical `x`, the functional equation is used to shift the argument to the range `0 < x < 1`. Cf. examples 3 and 4.

Example 1. We demonstrate some calls with exact and symbolic input data:

```
>> psi(-3/2), psi(4, 1), psi(3/2, 2)

                                 2
                               PI
       8/3 - 2 ln(2) - EULER, --- - 49/36, 16 - 14 zeta(3)
                                6

>> psi(x + sqrt(2), 4), psi(infinity, 5)
```

```
                      1/2
           psi(x + 2    , 4), 0
```

Floating point values are computed for floating point arguments:

```
>> psi(-5.2), psi(1.0, 3), psi(2.0 + 3.0*I, 10)

     6.065773152, 6.493939402, 0.7526409593 - 2.299472238 I
```

Example 2. `psi` is singular for nonpositive integers:

```
>> psi(-2)

 Error: singularity [psi]
```

Example 3. For positive integers and rational numbers x with denominators 2, 3, 4 and 6, respectively, the result is expressed in terms of `EULER`, `PI`, `ln`, and `zeta` if $|x| < 500$:

```
>> psi(-5/2), psi(-3/2, 1), psi(13/3, 2), psi(11/6, 4)

                           2
                         PI
 46/15 - 2 ln(2) - EULER, --- + 40/9,
                          2

                          3  1/2
                      4 PI  3
    75535713/1372000 - ---------- - 26 zeta(3),
                           9

          5  1/2
    176 PI  3    - 90024 zeta(5) + 186624/3125
```

For larger arguments, the `expand` attribute can be used to obtain such expressions:

```
>> psi(1000, 1)

                          psi(1000, 1)

>> expand(%)

   2
 PI
 --- -
  6

 835458876624295851523752364295.../508207201043258126178352 92...
```

Example 4. The functions `diff`, `expand`, `float`, `limit`, and `series` handle expressions involving `psi`:

```
>> diff(psi(x^2 + 1, 3), x), float(ln(3 + psi(sqrt(PI))))

                          2
                2 x psi(x  + 1, 4), 1.183103343

>> expand(psi(15/7))

                         psi(1/7) + 63/8

>> expand(psi(2*x + 3, 2))

  psi(x, 2)     1         2             2       psi(x + 1/2, 2)
  --------- + ---- + ---------- + ---------- + ---------------
      8          3            3             3          8
              4 x    (2 x + 1)    (2 x + 2)

>> limit(x*psi(x), x = 0), limit(psi(x, 3), x = infinity)

                              -1, 0

>> series(psi(x), x = 0), series(psi(x, 3), x = infinity, 3)

                   2                  3   4
   1            x PI     2           x   PI     4             5
 - - - EULER + ----- - x  zeta(3) + ------ - x  zeta(5) + O(x ),
   x              6                   90

    2    3    2    /  1 \
   -- + -- + -- + O| -- |
    3    4    5    |  6 |
   x    x    x     \ x  /
```

Changes:

- Special formulas for rational numbers with denominators 1, 2, 3, 4 or 6, respectively, were implemented.
- The system function `expand` now implements the reflection formula and the Gauß multiplication formula.

`quit` – terminate the MuPAD session

On the interactive level, the statement `quit` terminates the MuPAD session.

Call(s):

- quit
- _quit()

Related Functions: break, next, Pref::callOnExit, reset, return

---

Details:

- The quit statement is equivalent to the call _quit().
- If quit is used on the interactive level, it terminates the running MuPAD session and returns to the system level where MuPAD was started.
- quit should not be used in a procedure. However, if it is used, only this procedure is terminated. Note that in this case the return value of the procedure is undefined. Use return to terminate a procedure.
- When using a non-terminal version of MuPAD such as the MuPAD Pro Notebook, the Apple Macintosh user interfaces or the X11 user interfaces, the corresponding Quit button of the MuPAD session window must be used rather than the quit statement. In these versions, the quit statement leads to an error message.
- When a MuPAD session is terminated, so-called exit handlers are executed before exiting the MuPAD kernel. Exit handlers can be installed via the function Pref::callOnExit.
- _quit is a function of the system kernel.

Example 1. In this example, the Linux/UNIX terminal version of MuPAD is started and then terminated using the quit statement:

```
myprompt> mupad

   *----*    MuPAD 2.0.0 -- The Open Computer Algebra System
  /|   /|
 *----* |    Copyright (c)  1997 - 2000  by SciFace Software
 | *--|-*                   All rights reserved.
 |/   |/
 *----*      Universität Paderborn, FB-17, Mathematik

>> quit
myprompt>
```

Example 2. In a MuPAD version with a graphical user interface, e.g., under Windows 9x/NT/2000, the Apple Macintosh operating system, or Linux/UNIX with X11/Motif, a `quit` command results in the following error message:

```
>> quit

 Warning: Quit the kernel via the user interface [quit]
```

`radsimp` – simplify radicals in arithmetical expressions

`radsimp` simplifies arithmetical expressions containing radicals.

Call(s):

- `radsimp(z)`

Parameters:

`z` — an arithmetical expression

Return Value: an arithmetical expression.

Further Documentation: Chapter "Manipulating Expressions" of the Tutorial.

Related Functions: `combine`, `ifactor`, `normal`, `rectform`, `simplify`

Details:

- `radsimp(z)` tries to simplify the radicals in the expression `z`. The result is mathematically equivalent to `z`.
- The call `radsimp(z)` is equivalent to `simplify(z, sqrt)`.

Example 1. We demonstrate the simplification of constant expressions with square roots and higher order radicals:

```
>> radsimp(3*sqrt(7)/(sqrt(7) - 2)),
   radsimp(sqrt(5 + 2*sqrt(6)));
   radsimp(sqrt(5*sqrt(3) + 6*sqrt(2))),
   radsimp(sqrt(3 + 2*sqrt(2)))
```

```
          1/2        1/2    1/2
       2 7    + 7, 2    + 3

       3/4    1/2  1/4   1/2
      3    + 2    3   , 2    + 1
```

```
>> radsimp((1/2 + 1/4*3^(1/2))^(1/2))
```

```
     1/2    1/2  1/2
    2      2    3
    ---- + ---------
     4        4
```

```
>> radsimp((5^(1/3) - 4^(1/3))^(1/2))
```

```
     2/3    2/3    1/3  1/3
    4      5      4    5
    ---- - ---- + ---------
     6      3        3
```

```
>> radsimp(sqrt(3*sqrt(3 + 2*sqrt(5 - 12*sqrt(3 - 2*sqrt(2))))
                + 14))
```

```
     1/2
    2    + 3
```

```
>> radsimp(2*2^(1/4) + 2^(3/4) - (6*2^(1/2) + 8)^(1/2))
```

```
    0
```

```
>> radsimp(sqrt(1 + sqrt(3)) + sqrt(3 + 3*sqrt(3))
                - sqrt(10 + 6*sqrt(3)))
```

```
    0
```

Example 2. In some cases, you get the best result by using `radsimp` in combination with `simplify` :

```
>> radsimp(sqrt(9^(1/3) + 6*3^(1/3) + 9)); simplify(%)
```

```
     2/3
    9
    ---- + 3
     3

     1/3
    3    + 3
```

Example 3.

```
>> x := sqrt(3)*I/2 + 1/2: y := x^(1/3) + x^(-1/3): z := y^3 - 3*y

 /          1                     1/2       1/3 \3
 | --------------------- + (1/2 I 3    + 1/2)    |  -
 |         1/2       1/3                         |
 \ (1/2 I 3    + 1/2)                            /

             1/2       1/3              3
   3 (1/2 I 3    + 1/2)    - ---------------------
                                   1/2       1/3
                           (1/2 I 3    + 1/2)
```

```
>> radsimp(z)

                                 1
```

```
>> delete x, y, z:
```

Example 4. `radsimp` also works on arithmetical expressions containing variables:

```
>> z := x/(sqrt(3) - 1) - x/2

                                      x      x
                                  -------- - -
                                   1/2       2
                                  3    - 1
```

```
>> radsimp(z) = expand(radsimp(z))

                            /  1/2        \          1/2
                            | 3           |   x   x 3
                          x | ---- + 1/2  | - - = ------
                            \   2         /   2      2
```

```
>> delete z:
```

Background:

- For constant algebraic expressions, `radsimp` constructs a tower of algebraic extensions of $\mathbb{Q}$ using the domain `Dom::AlgebraicExtension`. It tries to return the simplest possible form.

- This function is based on an algorithm described in Borodin, Fagin, Hopcroft and Tompa, “Decreasing the Nesting Depth of Expressions Involving Square Roots”, JSC 1, 1985, pp. 169-188.
  In some special cases, an algorithm based on Landau, ”How to tangle with a nested radical”, The Mathematical Intelligencer 16, 1994, no. 2, pp. 49-55, is used.

`random` – generate random integer numbers

`random()` returns a random integer number between 0 and 10^{12}.

`random(n1..n2)` returns a procedure that generates random integers between `n1` and `n2`.

Call(s):

- `random()`
- `random(n1..n2)`
- `random(n)`

Parameters:

`n1, n2`	—	integers with `n1` $\leq$ `n2`
`n`	—	a positive integer

Return Value: `random()` returns a nonnegative integer. The calls `random(n1..n2)` and `random(n)` return a procedure of type `DOM_PROC`.

Side Effects: `random` as well as the random number generators created by it are sensitive to the environment variable `SEED`.

Related Functions: `frandom`, `stats::uniformRandom`

Details:

- The calls `random()` return uniformly distributed random integers between 0 and 999999999988 ($\approx 10^{12}$).
- `r := random(n1..n2)` produces a random number generator `r`. Subsequent calls `r()` generate uniformly distributed random integers between `n1` and `n2`.
- `random(n)` is equivalent to `random(0, n - 1)`.

- The global variable `SEED` is used for initializing or changing the sequence of random numbers. It may be assigned any *nonzero* integer. The value of `SEED` fixes the sequence of random numbers. This may be used to reset random generators and reproduce random sequences.

  `SEED` is set to a default value when MuPAD is initialized. Thus, each time MuPAD is started or re-initialized with the `reset` function, the random generators produce the same sequence of numbers.

- Several random generators produced by `random` may run simultaneously. All generators make use of the same global variable `SEED`.

- For producing uniformly distributed floating points numbers, it is recommended to use the faster function `frandom` instead. The `stats` library provides random generators with various other distributions. Cf. example 4.

Example 1. The following call produces a sequence of random integers. Note that an index variable `i` must be used in the construction of the sequence. A call such as `random() $ 8` would produce 8 copies of the same random value:

```
>> random() $ i = 1..8

 427419669081, 321110693270, 343633073697, 474256143563,

    558458718976, 746753830538, 32062222085, 722974121768
```

The following call produces a "die" that is rolled 20 times:

```
>> die := random(1..6): die() $ i = 1..20

   2, 2, 2, 4, 4, 3, 3, 2, 1, 4, 4, 6, 1, 1, 1, 2, 4, 2, 1, 3
```

The following call produces a "coin" that produces "head" or "tail":

```
>> coin := random(2): coin() $ i = 1..10

                  1, 0, 1, 1, 0, 1, 0, 1, 0, 0

>> subs(%, [0 = head, 1 = tail])

   tail, head, tail, tail, head, tail, head, tail, head, head

>> delete dice, coin:
```

Example 2. `random` is sensitive to the global variable `SEED` which is set and reset when MuPAD is (re-)initialized. The seed may also be set by the user. Random sequences can be reproduced by starting with a fixed `SEED`:

```
>> SEED := 1: random() $ i = 1..4

     427419669081, 321110693270, 343633073697, 474256143563

>> SEED := 1: random() $ i = 1..4

     427419669081, 321110693270, 343633073697, 474256143563
```

Example 3. `random` allows to create several random number generators for different ranges of numbers, and to use them simultaneously:

```
>> r1 := random(0..4): r2 := random(2..9): [r1(), r2()] $ i = 1..6

          [1, 4], [0, 2], [1, 3], [0, 5], [2, 2], [4, 7]

>> delete r1, r2:
```

Example 4. `random` can be used to build a random generator for uniformly distributed floating point numbers. The following generator produces such numbers between `-1.0` and `1.0`:

```
>> r := float@random(-10^DIGITS..10^DIGITS)/10^DIGITS:
   r() $ i = 1..12;

 0.2920457876, 0.3747019439, -0.5968604725, -0.9375052697,

    0.1053530039, -0.3513692809, 0.5590763459, -0.0607326312,

    -0.4571489053, 0.2600608968, 0.9760099364, 0.5982933733
```

However, it is strongly recommended to use the much more efficient functions `frandom` or `stats::uniformRandom` instead:

```
>> r := stats::uniformRandom(-1, 1, Seed = 10^10):
   r() $ i = 1..12

 -0.06841411958, 0.2765669032, 0.2567349114, 0.7462262409,

    -0.05021280233, -0.8537725277, 0.9460955434, -0.4278325857,

    0.2170908991, 0.3648317893, 0.7506129998, 0.6918057213

>> delete r:
```

Background:

- `random` implements a linear congruence generator. The sequence of pseudo-random numbers generated by calling `random()` over and over again is $f(x), f(f(x)), \ldots$, where x is the initial value of `SEED` and f is the function $x \mapsto a\,x \bmod m$ with suitable integer constants a and m.

`rationalize` – transform an expression into a rational expression

`rationalize(object)` transforms the expression `object` into an equivalent rational expression by replacing non-rational subexpressions by newly generated variables.

Call(s):

- `rationalize(object, <, inspect <, stop>>)`

Parameters:

`object`	—	an arithmetical expression or a set or list of such expressions
`inspect`	—	subexpressions to operate on: a set of types, or a procedure, or `NIL`. The default is `NIL`, i.e., *all* subexpressions are to be inspected.
`stop`	—	subexpressions to be leaft unchanged: a set of types, or a procedure, or `NIL`. The default is the set `{DOM_INT`, `DOM_RAT`, `DOM_IDENT}`, i.e., integers, rational numbers and identifiers are not replaced by variables.

Return Value: a sequence consisting of the rationalized object and a set of substitution equations.

Related Functions: `indets`, `maprat`, `rewrite`, `simplify`, `subs`

Details:

- An expression or a subexpression is regarded as "non-rational", if it is neither a sum, nor a product, nor a power with an integer exponent.

  `rationalize(object, inspect, stop)` "walks" recursively through the expression tree of `object` as long as the types of the subexpressions are in `inspect`. All non-rational subexpressions of a type not matching `stop` are replaced by variables `D1`, `D2`, etc.

- `rationalize` returns a sequence `(rat, subsSet)`. The rationalized object `rat` contains new variables, which are specified by the set of "substitution equations" `subsSet`. The relation `object = subs(rat, subsSet)` holds.

- If `inspect` is `NIL`, all subexpressions are inspected. If `inspect` is a set of types, all subexpressions matching one of these types are inspected. If `inspect` is a procedure, all subexpressions `x`, say, with `inspect(x) = TRUE` are inspected.

  Any subexpression not matching `inspect` is replaced by a variable.

- If `stop` is `NIL`, then all inspected non-rational subexpressions are replaced by variables. If `stop` is a set of types, any non-rational subexpression matching one of these types is left untouched. If `stop` is a procedure, any non-rational subexpression `x`, say, with `stop(x) = TRUE` is leaft untouched.

- The types in `inspect` and `stop` may be strings as returned by the `type` function, or domain types such as `DOM_INT`, `DOM_RAT` etc.

Example 1. `rationalize` operates on single arithmetical expressions as well as on lists and sets of expressions:

```
>> rationalize(2*sqrt(3) + 0.5*x^3)

                        3          1/2
              2 D2 + D1 x , {D2 = 3   , D1 = 0.5}

>> rationalize([(x - sqrt(2))*(x^2 + sqrt(3)),
                (x - sqrt(2))*(x - sqrt(3))])

                 2                              1/2         1/2
 [(x - D3) (D4 + x ), (x - D3) (x - D4)], {D3 = 2   , D4 = 3   }
```

Example 2. `rationalize` allows to specify which kinds of subexpressions are to be inspected and which kinds of subexpressions are to be leaft unchanged. In the following call, the subexpression `x^3` (of type `"_power"`) is not inspected and replaced by a variable:

```
>> rationalize(2*sqrt(3) + 0.5*x^3, {"_plus", "_mult"})

                              3         1/2
          2 D5 + D6 D7, {D6 = x , D5 = 3   , D7 = 0.5}
```

In the following call, all subexpressions are inspected. Neither floating point numbers nor integers nor identifiers are replaced:

```
>> rationalize(2*sqrt(3) + 0.5*x^3, NIL,
               {DOM_FLOAT, DOM_INT, DOM_IDENT})

                              3          1/2
                   2 D8 + 0.5 x , {D8 = 3   }
```

`read` – search, read, and execute a file

`read(filename)` searches for the file `filename` in certain directories, reads and executes it.

`read(n)` reads and executes the file associated with the file descriptor `n`.

Call(s):

- `read(filename <, Quiet> <, Plain>)`
- `read(n <, Quiet> <, Plain>)`

Parameters:

`filename`	—	the name of a file: a character string
`n`	—	a file descriptor provided by `fopen`: a positive integer

Options:

`Plain`	—	makes `read` use its own parser context
`Quiet`	—	suppresses output during execution of `read`

Return Value: the return value of the last statement of the file.

Related Functions: `fclose`, `fileIO`, `FILEPATH`, `finput`, `fname`, `fopen`, `fprint`, `fread`, `ftextinput`, `input`, `LIBPATH`, `loadproc`, `pathname`, `print`, `protocol`, `READPATH`, `textinput`, `write`, `WRITEPATH`

Details:

- `read(filename)` searches for the file in various directories:

  - First, the name is interpreted as a relative file name: `filename` is concatenated to each directory given by the environment variable `READPATH`.
  - Then the file name is interpreted as an absolute path name.
  - Then the file name is interpreted relative to the "working directory".
  - Last, the file name is concatenated to each directory given by the environment variable `LIBPATH`.

  If a file can be opened with one of this names, then the file is read and executed with `fread`.

- Please note that the "working directory", which is used to interpret relative file names, depends on the operating system. On Windows systems, the "working directory" is the folder, where MuPAD is installed. On UNIX or Linux systems, it is the directory where MuPAD was started.
- A path separator ("`/`" on UNIX or Linux, "\" on Windows and "`:`" on the Macintosh) is inserted as neccessary when concatenating a given path and `filename`.
- On the Macintosh, an empty file name may be given. In this case a dialogue box is opened in which the user can choose a file.
- `read(n)` with a file descriptor `n` as returned by `fopen` is equivalent to the call `fread(n)`.
- See the function `fread` for details about reading and executing the file's content and for a detailed description of the options *Plain* and *Quiet*.
- When a file is read with `read` the variable `FILEPATH` contains the path of the file.
- For an overview of all file related MuPAD functions, also try `?fileIO`.

Example 1. The following example only works under UNIX and Linux; on other operating systems one must change the path names accordingly. First, we use `write` to store values in the file "`testfile.mb`" in the "`/tmp`" directory:

```
>> a := 3: b := 5: write("/tmp/testfile.mb", a, b):
```

The following command specifies the file by its absolute path name. After reading the file, the values of `a` and `b` are restored:

```
>> delete a, b: read("/tmp/testfile.mb"): a, b

                              3, 5
```

Alternatively, we define "`/tmp`" as the search directory and provide a relative path name. Note that the path separator "`/`" is inserted by `read`:

```
>> delete a, b: READPATH := "/tmp": read("testfile.mb"): a, b

                              3, 5
```

We may also use `fopen` to open the file and read its content. Note that `fopen` does not search for the file like `read`, thus we must enter an absolute path name or a name relative to the working directory:

```
>> delete a, b:
   n := fopen("/tmp/testfile.mb"): read(n): fclose(n):
   a, b
```

```
                    3, 5

>> delete a, b, READPATH, n
```

---

readbytes, writebytes – read or write binary data from or to a file

`readbytes(filename)` reads the binary file named `filename`. The data in the file are returned as a list of numbers.

`readbytes(n)` reads the file associated with the file descriptor `n`.

`writebytes(filename, list)` writes a list of MuPAD numbers as a stream of binary data to the file `filename`.

`writebytes(n, list)` writes the contents of the list to the file associated with the file descriptor `n`.

Call(s):

- readbytes(filename <, m> <, format> <, byteorder>)
- readbytes(n <, m> <, format> <, byteorder>)
- writebytes(filename, list <, format> <, byteorder>)
- writebytes(n, list <, m> <, format> <, byteorder>)

Parameters:

`filename`	—	the name of a file: a character string or the flag *TempFile*
`n`	—	a file descriptor provided by `fopen`: a positive integer. The file must have been be opened using the `fopen`-flag *Raw*.
`list`	—	a list of MuPAD numbers that are to be written to the file. The entries must match the specified `format`.
`m`	—	the number of values to be read or written: a positive integer.

Options:

`format`	—	the format of the binary data: either *Byte*, *SignedByte*, *Short*, *SignedShort*, *Word*, *SignedWord*, *Float* or *Double*. The default format is *Byte*.
`byteorder`	—	the byte ordering: either *BigEndian* or *LittleEndian*. The default odering is *BigEndian*.

Return Value: `readbytes` returns a list of MuPAD numbers (either integers or floating point numbers); `writebytes` returns the void object `null()` of type `DOM_NULL`.

Side Effects: The function `readbytes` is sensitive to the environment variable `READPATH`. First, the file is searched in the "working directory". If it cannot be found there, all paths in `READPATH` are searched.

The function `writebytes` is sensitive to the environment variable `WRITEPATH`. If this variable has a value, the file is created in the corresponding directory. Otherwise, the file is created in the "working directory."

Related Functions: `fclose`, `fileIO`, `FILEPATH`, `finput`, `fname`, `fopen`, `fprint`, `fread`, `ftextinput`, `pathname`, `print`, `protocol`, `read`, `READPATH`, `write`, `WRITEPATH`

Details:

- `readbytes` and `writebytes` are MuPAD functions that enable the user to read or write arbitrary files and interpret their contents as a sequence of numbers.

- These functions are particularly useful to work on data provided by or destined for external programs. You can use them, for example, to implement encryption or compression algorithms in MuPAD. Cf. example 2.

- The results of `readbytes` and `writebytes` depend on the interpretation of the binary data set by the `format` option. When reading/writing a file, you can interpret it as a stream of *Byte*, *SignedByte*, *Short*, *SignedShort*, *Word*, *SignedWord*, *Float* or *Double*. These are standard formats that are used by many program packages to write/read data. Cf. example 1.

  Be sure to read/write the data in the appropriate way. You need to know the format used by the program which created the file or is supposed to read the file, respectively.

- When writing data via `writebytes`, each entry in the list is checked for whether it can be converted to the specified format. If this is not the case, `writebytes` raises an error. Cf. example 4.

- The file may be specified directly by its name. In this case, `writebytes` creates a new file or overwrites an existing file. See the help page of `fopen` for the special flag *TempFile*.

  If a file name is specified, `readbytes` and `writebytes` open and close the file automatically.

  If `WRITEPATH` or `READPATH` have no value, `writebytes` and `readbytes` interpret the file name as a pathname relative to the "working directory."

  Note that the meaning of "working directory" depends on the operating system. On Windows systems, the "working directory" is the folder where MuPAD is installed. On UNIX or Linux systems, it is the current working directory in which MuPAD was started.

On the Macintosh, an empty file name may be given. In this case, a dialog box is opened in which the user can choose a file. Further, on the interactive level, MacMuPAD warns the user if an existing file is about to be overwritten.

Absolute path names are processed by `readbytes` and `writebytes`, too.

- If a file name is specified, each call to `readbytes`/`writebytes` opens the file at the beginning. If the file was opened via `fopen`, subsequent calls of `readbytes` or `writebytes` with the corresponding file descriptor start at the point in the file that was reached by the last `readbytes`/`writebytes` command.

  Hence, if you want to read/write a file by portions, you must open it with `fopen` and use the returned file descriptor instead of the filename. Cf. example 3.

  If the file is to be opened via `fopen`, be sure to pass the flag *Raw* to `fopen`. Otherwise, `readbytes` and `writebytes` raise an error.

- If the number of bytes in the file in a `readbytes` call is not a multiple of units of the specified format, the data are read up to the last complete number. The remaining bytes are ignored. Cf. example 5.

- For an overview of all file related MuPAD functions, also try `?fileIO`.

- `writebytes` is a function of the system kernel.

Option <`Byte, SignedByte`>**:**

- A byte is an 8-bit binary number. Therefore, a byte can have 2^8 different values. For *Byte*, these are the integers from 0 to 255. For *SignedByte*, they are the integers from -128 to 127.

- With *Byte*, the data are read/written in 8-bit blocks, interpreted as unsigned bytes. When writing, the numbers are checked for being in the range from 0 to 255.

- With *SignedByte*, the data are read or written using the 2-complement.

- *Byte* is the default format.

- Cf. example 1 for an overview over the different format-options.

Option <`Short, SignedShort`>**:**

- A 'short' is a 16-bit binary number (2 bytes). Therefore, a 'short' can have 2^{16} different values. For *Short*, these are the integers from 0 to 65536. For *SignedShort*, they are the integers from -32768 to 32767.

- Cf. the flags *BigEndian* and *LittleEndian* for details on the byte ordering.
- The semantics of *Short* or *SignedShort* is analogeous to that of *Byte* or *SignedByte*, respectively.

Option <`Word, SignedWord`>**:**

- A 'word' is a 32-bit binary number (4 bytes). Therefore, a 'word' can have 2^{32} different values. For *Word*, these are the integers from 0 to 4294967296. For *SignedWord*, they are the integers from -2147483648 to 2147483647.
- Cf. the flags *BigEndian* and *LittleEndian* for details on the byte ordering.
- The semantics of *Word* or *SignedWord* is analogeous to that of *Byte* or *SignedByte*, respectively.

Option <`Float, Double`>**:**

- A 'float' is a 32-bit representation of a real number (4 bytes). A 'double' is a 64-bit representation of a real number (8 bytes).

  Floats and doubles are read/written in the format of the machine/operating system MuPAD is currently running on. Therefore, the results may differ between different platforms.

  Binary files containig floating point numbers are, in general, not portable to other platforms.
- Cf. the flags *BigEndian* and *LittleEndian* for details on the byte ordering.
- The semantics of *Float* or *Double* is analogeous to that of *Byte* or *SignedByte*, respectively.

Option <`BigEndian, LittleEndian`>**:**

- *BigEndian* and *LittleEndian* specify the order in which the bytes are arranged for *Short*, *SignedShort*, *Word*, *SignedWord*, *Float*, and *Double*.

- For all formats, the data are written in 8-bit blocks (bytes). This also includes the formats where a unit is longer than one byte (all formats but *Byte* and *SignedByte*). With *BigEndian*, the bytes with the most significant bits ('high bits') are written first. With *LittleEndian*, the bytes with the least significant bits are written first.

  If, for example, *Short* is selected, there are 16 bits that are to be written. If you pass *BigEndian*, first the byte with the bits for 2^{15} to 2^8 and then the byte with the bits for 2^7 to 2^0 are written. If you specify *LittleEndian*, the order of the bytes is reversed.

- *BigEndian* and *LittleEndian* have no effect if the formats *Byte* or *SignedByte* are specified.

- *BigEndian* is the default byte order.

- Cf. example 6 for the effects of *BigEndian* and *LittleEndian*.

Example 1. In this example, we write a sequence of numbers to the file `test.tst` with the default settings. Then, we load them back in:

```
>> writebytes("test.tst", [42, 17, 1, 3, 5, 7, 127, 250]):

>> readbytes("test.tst")

                  [42, 17, 1, 3, 5, 7, 127, 250]
```

We now read the above data with some other option: *SignedByte* interprets all values from 0 to 127 exactly as *Byte* does. Higher values x, however, are interpreted as $x - 256$. For example, $250 - 256 = -6$:

```
>> readbytes("test.tst", SignedByte)

                  [42, 17, 1, 3, 5, 7, 127, -6]
```

Short interprets two bytes to be one number. Therefore, the eight written bytes are interpreted as four numbers. For example, the first 2 bytes yield $42 \cdot 2^8 + 17 = 10769$:

```
>> readbytes("test.tst", Short)

                  [10769, 259, 1287, 32762]
```

With the flag *LittleEndian*, the byte order is reversed. For example, the first 2 bytes now yield $17 \cdot 2^8 + 42 = 4394$:

```
>> readbytes("test.tst", Short, LittleEndian)

                  [4394, 769, 1797, 64127]
```

Word interprets four bytes to be one number. Therefore, the eight written bytes give two numbers. The first 4 bytes yield $10769 \cdot 2^{16} + 259 = 705757443$:

```
>> readbytes("test.tst", Word)
```

```
[705757443, 84377594]
```

Double interprets eight bytes to represent one floating-point number. The interpretation is machine dependent and may be different for you:

```
>> readbytes("test.tst", Double)
```

```
[4.633737352e-106]
```

Example 2. We use `readbytes` and `writebytes` to encrypt the file created in the previous example with a simple 'CAESAR type encoding': Any integer x (a byte) is replaced by $x + 13 \mod 256$:

```
>> L := readbytes("test.tst"):
   L := map(L, x -> (x + 13 mod 256)):
   writebytes("test.tst", L):
```

Knowing the encryption, we can successfully decrypt the file:

```
>> L := readbytes("test.tst")
```

```
[55, 30, 14, 16, 18, 20, 140, 7]
```

```
>> map(L, x -> (x - 13 mod 256))
```

```
[42, 17, 1, 3, 5, 7, 127, 250]
```

```
>> delete L:
```

Example 3. In this example, we use `fopen` to write and read a file in portions:

```
>> n := fopen("test.tst", Write, Raw):
   for i from 1 to 10 do writebytes(n, [i]) end_for:
   fclose(n):
```

Equivalently, we could have written all data in one go:

```
>> n := fopen("test.tst", Write, Raw):
   writebytes(n, [i $ i = 1..10]):
   fclose(n):
```

We read the data byte by byte:

```
>> n := fopen("test.tst", Read, Raw):
   readbytes(n, 1), readbytes(n, 1), readbytes(n, 1);
   fclose(n):
```

```
[1], [2], [3]
```

The next command reads in portions of 5 bytes each:

```
>> n := fopen("test.tst", Read, Raw):
   readbytes(n, 5), readbytes(n, 5);
   fclose(n):
```

```
[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]
```

```
>> delete n, i:
```

Example 4. An error is raised if the data do not match the specified format. Here, `-5` does not match *Byte*. This format does not include negative numbers:

```
>> writebytes("test.tst", [42, 17, -5, 7], Byte)
```

```
Error: Illegal argument [writebytes]
```

Example 5. Here we demonstrate what happens if the number of bytes in the file does not match a multiple of units of the specified format. Since both *SignedShort* and *Float* consist of numbers of 2 bytes each, the trailing 5-th byte corresponding to `11` is ignored:

```
>> writebytes("test.tst", [42, 17, 7, 9, 11], Byte):
   readbytes("test.tst", SignedShort),
   readbytes("test.tst", Float)
```

```
[10769, 1801], [1.28810279e-13]
```

Example 6. Here we show the effects of *BigEndian* and *LittleEndian*:

```
>> writebytes("test.tst", [129, 255, 145, 171, 191, 253], Byte):
   L1 := readbytes("test.tst", Short, BigEndian)
```

```
[33279, 37291, 49149]
```

```
>> L2 := readbytes("test.tst", Short, LittleEndian)
```

```
[65409, 43921, 64959]
```

We look at the data in a binary representation (see `numlib::g_adic` for details). The effect of using *`LittleEndian`* instead of *`BigEndian`* is to exchange the first 8 bits and the last 8 bits of each number:

```
>> map(L1, numlib::g_adic, 2)
```

```
 [[1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1],

    [1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1],

    [1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1]]
```

```
>> map(L2, numlib::g_adic, 2)
```

```
 [[1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1],

    [1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1],

    [1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1]]
```

```
>> delete L1, L2:
```

Changes:

- `readbytes` and `writebytes` are new functions.

`repeat, while` – repeat and while loop

`repeat - end_repeat` is a loop that evaluates its body until a specified stopping criterion is satisfied.

`while - end_while` represents a loop that evaluates its body while a specified condition holds true.

Call(s):

- ```
  repeat
    body
  until condition end_repeat
  ```
- `_repeat(body, condition)`

- ```
  while condition do
    body
  end_while
  ```
- `_while(condition, body)`

Parameters:

`body`	—	the body of the loop: an arbitrary sequence of statements
`condition`	—	a Boolean expression

Return Value: the value of the last command executed in the body of the loop. If no command was executed, the value `NIL` is returned. If the body of a `while` loop is not evaluated due to a false condition, the void object of type `DOM_NULL` is returned.

Further Documentation: Chapter 16 of the MuPAD Tutorial.

Related Functions: `break`, `for`, `next`, `_lazy_and`, `_lazy_or`

Details:

- In a `repeat` loop, first `body` and then `condition` are evaluated until `condition` evaluates to `TRUE`.
- In a `while` loop, `condition` is evaluated before the body is executed for the first time. If `condition` evaluates to `TRUE`, the loop is entered and `body` and `condition` are evaluated until `condition` evaluates to `FALSE`.
- In contrast to the `while` loop, the body of a `repeat` loop is always evaluated at least once.
- The body may consist of any number of statements which must be separated either by a colon `:` or a semicolon `;`. Only the last evaluated result inside the body (the return value of the loop) is printed on the screen. Use `print` to see intermediate results.
- The Boolean expression `condition` must be reducible to either `TRUE` or `FALSE`. Internally, the condition is evaluated in the lazy evaluation context of the functions `_lazy_and` and `_lazy_or`.
- The statements `next` and `break` can be used in `repeat` and `while` loops in the same way as in `for` loops.
- The keywords `end_repeat` and `end_while` may be replaced by the keyword `end`.
- The imperative forms `repeat - end_repeat` and `while - end_while` are equivalent to corresponding calls of the functions `_repeat` and `_while`, respectively. In most cases, the imperative forms should be preferred because they lead to simpler code.
- `_repeat` and `_while` are functions of the system kernel.

Example 1. Intermediate results of statements within a `repeat` and `while` loop are not printed to the screen:

```
>> i := 1:
   s := 0:
   while i < 3 do
     s := s + i;
     i := i + 1;
   end_while
```

$$3$$

Above, only the return value of the loop is displayed. Use `print` to see intermediate results:

```
>> i := 1:
   s := 0:
   while i < 3 do
     print("intermediate sum" = s);
     s := s + i;
     i := i + 1;
     s
   end_while
```

```
"intermediate sum" = 0

"intermediate sum" = 1

3
```

```
>> delete i, s:
```

Example 2. A simple example is given, how a `repeat` loop can be expressed via an equivalent `while` loop. For other examples, this may be more complicated and additional initializations of variables may be needed:

```
>> i := 1:
   repeat
     print(i);
     i := i + 1;
   until i = 3 end:
```

```
1

2
```

```
>> i := 1:
   while i < 3 do
     print(i);
     i := i + 1;
   end:
```

```
1

2
```

```
>> delete i:
```

Example 3. The Boolean expression `condition` must evaluate to `TRUE` or `FALSE`:

```
>> condition := UNKNOWN:
   while not condition do
     print(Condition = condition);
     condition := TRUE;
   end_while:
```

```
 Error: Unexpected boolean UNKNOWN [while]
```

To avoid this error, change the stopping criterion to `condition <> TRUE`:

```
>> condition := UNKNOWN:
   while condition <> TRUE do
     print(Condition = condition);
     condition := TRUE;
   end_while:
```

```
Condition = UNKNOWN
```

```
>> delete condition:
```

Example 4. We demonstrate the correspondence between the functional and the imperative form of the `repeat` and `while` loop, respectively:

```
>> hold(_repeat((statement1; statement2), condition))
```

```
repeat
  statement1;
  statement2
until condition end_repeat
```

```
>> hold(_while(condition, (statement1; statement2)))
```

```
while condition do
  statement1;
  statement2
end_while
```

`rec` – the domain of recurrence equations

`rec(eq, y(n))` represents a recurrence equation for the sequence `y(n)`.

Call(s):

- `rec(eq, y(n)` <, `cond`>)

Parameters:

`eq`	—	an equation or an arithmetical expression
`y`	—	the unknown function: an identifier
`n`	—	the index: an identifier
`cond`	—	a set of initial or boundary conditions

Return Value: an object of type `rec`.

Related Functions: `ode`, `solve`, `sum`

Details:

- `rec(eq, y(n))` creates an object of type `rec` representing a recurrence equation for `y(n)`.

  The equation `eq` must involve only shifts `y(n + i)` with integer values of `i`; at least one such expression must be present in `eq`. An arithmetical expression `eq` is equivalent to the equation `eq = 0`.

  Initial or boundary conditions `cond` must be specified as sets of equations of the form `{y(n0) = y0, y(n1) = y1, ...}` with arithmetical expressions `n0`, `n1`, ... that must not contain the identifier `n`, and arithmetical expressions `y0`, `y1`, ... that must not contain the identifier `y`.

- The main purpose of the `rec` domain is to provide an environment for overloading the function `solve`. For a recurrence `r` of type `rec`, the call `solve(r)` returns a set representing an affine subspace of the complete solution space. Its only entry is an expression in `n` that may contain free parameters such as `C1`, `C2` etc. Cf. the examples 1, 4, and 5.

- Currently only linear recurrences with coefficients that are rational functions of `n` can be solved. `solve` handles recurrences with constant coefficients, it finds hypergeometric solutions of first order recurrences, and polynomial solutions of higher order recurrences with non-constant coefficients.

- solve is not always able to find the complete solution space. Cf. example 5. If solve cannot find a solution, then the solve call is returned symbolically. For parametric recurrences, the output of solve may be a conditionally defined set of type piecewise. Cf. example 6.

Example 1. The first command defines the homogeneous first order recurrence equation $y(n+1) = 2\,(n+1)\,y(n)/n$ for the sequence $y(n)$. It is solved by a call to the solve function:

```
>> rec(y(n + 1) = 2*y(n)*(n + 1)/n, y(n))

             /             2 y(n) (n + 1)           \
          rec| y(n + 1) - --------------, y(n), {} |
             \                   n                  /
```

```
>> solve(%)

                                n
                         {n C1 2 }
```

Thus, the general solution of the recurrence equation is $y(n) = C_1\,n\,2^n$, where C_1 is an arbitrary constant.

Example 2. In the next example, the homogeneous first order recurrence $y(n+1) = 3\,(n+1)\,y(n)$ with the initial condition $y(0) = 1$ is solved for the unknown sequence $y(n)$:

```
>> solve(rec(y(n + 1) = 3*(n + 1)*y(n), y(n), {y(0) = 1}))

                         n
                       {3  gamma(n + 1)}
```

Thus, the solution is $y(n) = 3^n \cdot \Gamma(n+1) = 3^n \cdot n!$ for all integers $n \geq 0$ (Γ is the gamma function).

Example 3. In the following example, the inhomogeneous second order recurrence $y(n+2) - 2\,y(n+1) + y(n) = 2$ is solved for the unknown sequence $y(n)$. The initial conditions $y(0) = -1$ and $y(1) = m$ with some parameter m are taken into account by solve:

```
>> solve(rec(y(n + 2) - 2*y(n + 1) + y(n) = 2, y(n),
             {y(0) = -1, y(1) = m}))

                              2
                        {m n + n  - 1}
```

Example 4. We compute the general solution of the homogeneous second order recurrence $y(n+2) + 3\,y(n+1) + 2\,y(n) = 0$:

```
>> solve(rec(y(n + 2) + 3*y(n + 1) + 2*y(n), y(n)))

                             n          n
                 {C6 (-1)  + C7 (-2) }
```

Here, C6 and C7 are arbitrary constants.

Example 5. For the following homogeneous third order recurrence with non-constant coefficients, the system only finds the polynomial solutions:

```
>> solve(rec(n*y(n + 3) = (n + 3)*y(n), y(n)))

                                {n C9}
```

Example 6. The following homogeneous second order recurrence with constant coefficients involves a parameter a. The solution set depends on the value of this parameter, and solve returns a piecewise object:

```
>> solve(rec(a*y(n + 2) = y(n), y(n)))

          /              {     /   1  \n       /      1  \n }
piecewise| {0} if a = 0, { C11 | ---- |  + C10 | - ---- |  }
          |              {     |  1/2 |        |    1/2 |  }
          \              {     \ a    /        \   a    /  }

            \
   if a <> 0 |
             |
             /
```

Example 7. The following homogeneous second order recurrence with non-constant coefficients involves a parameter a. Although it has a polynomial solution for $a = 2$, the system does not recognize this:

```
>> solve(rec(n*y(n + 2) = (n + a)*y(n), y(n)))

                                 {0}
```

Background:

- For homogeneous recurrences with constant coefficients, `solve` computes the roots of the characteristic polynomial. If some of them cannot be given in explicit form, i.e., only by means of `RootOf`, then `solve` does not return a solution. Otherwise, the complete solution space is returned.
- For first order homogeneous recurrences with nonconstant coefficients, `solve` returns the complete solution space if the coefficients of the recurrence can be factored into at most quadratic polynomials. Otherwise, `solve` does not return a solution.
- For homogeneous recurrences of order at least two with nonconstant coefficients, `solve` finds the complete space of all *polynomial* solutions.
- Currently, inhomogeneous recurrences can only be solved if they have a polynomial solution. The previous remarks apply.
- For parametric recurrences, the system may not find solutions that are valid only for special values of the parameters. Cf. example 7.

`rectform` – rectangular form of a complex expression

`rectform(z)` computes the rectangular form of the complex expression z, i.e., it splits z into $z = \Re(z) + i\,\Im(z)$.

Call(s):

- `rectform(z)`

Parameters:

`z` — an arithmetical expression, a polynomial, a series expansion, an array, a list, or a set

Return Value: an element of the domain `rectform` if `z` is an arithmetical expression, and an object of the same type as `z` otherwise.

Side Effects: The function is sensitive to properties of identifiers set via `assume`; see example 3.

Overloadable by: `z`

Further Documentation: Chapter "Manipulating Expressions" of the Tutorial.

Related Functions: `abs`, `assume`, `collect`, `combine`, `conjugate`, `expand`, `Im`, `normal`, `radsimp`, `Re`, `rewrite`, `sign`, `simplify`

Details:

- `rectform(z)` tries to split `z` into its real and imaginary part and to return `z` in the form $\Re(z) + i\,\Im(z)$.

  `rectform` works recursively, i.e., it first tries to split each subexpression of `z` into its real and imaginary part and then tackles `z` as a whole.

- Use `Re` and `Im` to extract the real and imaginary parts, respectively, from the result of `rectform`. See example 1.

- `rectform` is more powerful than a direct application of `Re` and `Im` to `z`. However, usually it is much slower. For constant arithmetical expressions, it is therefore recommended to use the functions `Re` and `Im` directly. See example 2.

- The main use of `rectform` is for symbolic expressions, and properties of identifiers are taken into account (see `assume`). An identifier without any property is assumed to be complex valued. See example 3.

- If `z` is a array, a list, or a set, then `rectform` is applied to each entry of `z`.

  If `z` is a polynomial or a series expansion, of type `Series::Puiseux` or `Series::gseries`, then `rectform` is applied to each coefficient of `z`.

  See example 5.

- The result `r := rectform(z)` is an element of the domain `rectform`. Such a domain element consists of three operands, satisfying the following equality:
  `z = op(r, 1) + I*op(r, 2) + op(r, 3)`.
  The first two operands are real arithmetical expressions, and the third operand is an expression that cannot be splitted into its real and imaginary part.

  Sometimes `rectform` is unable to compute the required decomposition. Then it still tries to return some partial information by extracting as much as possible from the real and imaginary part of `z`. The extracted parts are stored in the first two operands, and the third operand contains the remainder, where no further extraction is possible. In extreme cases, the first two operands may even be zero. Example 6 illustrates some possible cases.

- Arithmetical operations with elements of the domain type `rectform` are possible. The result of an arithmetical operation is again an element of this domain (see example 4).

- Most MuPAD functions handling arithmetical expressions (e.g., `expand`, `normal`, `simplify` etc.) can be applied to elements of type `rectform`. They act on each of the three operands individually.

☞ Use `expr` to convert the result of `rectform` into an element of a basic domain; see example 4.

Example 1. The rectangular form of $\sin(z)$ for complex values z is:

```
>> delete z: r := rectform(sin(z))

       sin(Re(z)) cosh(Im(z)) + (cos(Re(z)) sinh(Im(z))) I
```

The real and the imaginary part can be extracted as follows:

```
>> Re(r), Im(r)

         sin(Re(z)) cosh(Im(z)), cos(Re(z)) sinh(Im(z))
```

The complex conjugate of `r` can be obtained directly:

```
>> conjugate(r)

       sin(Re(z)) cosh(Im(z)) - (cos(Re(z)) sinh(Im(z))) I
```

Example 2. The real and the imaginary part of a constant arithmetical expression can be determined by the functions `Re` and `Im`, as in the following example:

```
>> Re(ln(-4)) + I*Im(ln(-4))

                          I PI + ln(4)
```

In fact, they work much faster than `rectform`. However, they fail to compute the real and the imaginary part of arbitrary symbolic expressions, such as for the term $e^{i \sin z}$:

```
>> delete z: f := exp(I*sin(z)):
   Re(f), Im(f)

              Re(exp(I sin(z))), Im(exp(I sin(z)))
```

The function `rectform` is more powerful. It is able to split the expression above into its real and imaginary part:

```
>> r := rectform(f)

 cos(sin(Re(z)) cosh(Im(z))) exp(-cos(Re(z)) sinh(Im(z))) +

    (sin(sin(Re(z)) cosh(Im(z))) exp(-cos(Re(z)) sinh(Im(z)))) I
```

Now we can extract the real and the imaginary part of `f`:

```
>> Re(r)

    cos(sin(Re(z)) cosh(Im(z))) exp(-cos(Re(z)) sinh(Im(z)))

>> Im(r)

    sin(sin(Re(z)) cosh(Im(z))) exp(-cos(Re(z)) sinh(Im(z)))
```

Example 3. Identifiers without properties are considered to be complex variables:

```
>> delete z: rectform(ln(z))

                 2        2
         ln(Im(z)  + Re(z) )
         ------------------- + I arg(I Im(z) + Re(z))
                  2
```

However, you can affect the behavior of `rectform` by attaching properties to the identifiers. For example, if z assumes only real negative values, the real and the imaginary part simplify considerably:

```
>> assume(z < 0): rectform(ln(z))

                          ln(-z) + I PI
```

Example 4. We compute the rectangular form of the complex variable x:

```
>> delete x: a := rectform(x)

                         Re(x) + I Im(x)
```

Then we do the same for the real variable y:

```
>> delete y: assume(y, Type::Real): b := rectform(y)

                                y

>> domtype(a), domtype(b)

                       rectform, rectform
```

We have stored the results, i.e., the elements of domain type `rectform`, in the two identifiers `a` and `b`. We compute the sum of `a` and `b`, which is again of domain type `rectform`, i.e., it is already splitted into its real and imaginary part:

```
>> c := a + b
```

```
                  (y + Re(x)) + I Im(x)
```

```
>> domtype(c)
```

```
                        rectform
```

The result of an arithmetical operation between an element of domain type `rectform` and an arbitrary arithmetical expression is of domain type `rectform` as well:

```
>> delete z: d := a + 2*b + exp(z)
```

```
 (2 y + Re(x) + cos(Im(z)) exp(Re(z))) +

    I (Im(x) + sin(Im(z)) exp(Re(z)))
```

```
>> domtype(d)
```

```
                        rectform
```

Use the function `expr` to convert an element of domain type `rectform` into an element of a basic domain:

```
>> expr(d)
```

```
 2 y + I Im(x) + Re(x) + cos(Im(z)) exp(Re(z)) +

    I sin(Im(z)) exp(Re(z))
```

```
>> domtype(%)
```

```
                        DOM_EXPR
```

Example 5. `rectform` also works for polynomials and series expansions, namely individually on each coefficient:

```
>> delete x, y: p := poly(ln(-4) + y*x, [x]):
   rectform(p)
```

```
      poly((Re(y) + I Im(y)) x + (ln(4) + I PI), [x])
```

Similarly, `rectform` works for lists, sets, or arrays, where it is applied to each individual entry:

```
>> a := array(1..2, [x, y]):
   rectform(a)
```

```
+-                                 -+
| Re(x) + I Im(x), Re(y) + I Im(y) |
+-                                 -+
```

Note that `rectform` does not work directly for other basic data types. For example, if the input expression is a table of arithmetical expressions, then `rectform` responds with an error message:

```
>> a := table("1st" = x, "2nd" = y):
   rectform(a)

 Error: invalid argument, expecting an arithmetical expression \
 [rectform::new]
```

Use `map` to apply `rectform` to the operands of such an object:

```
>> map(a, rectform)

                    table(
                      "2nd" = Re(y) + I Im(y),
                      "1st" = Re(x) + I Im(x)
                    )
```

Example 6. This example illustrates the meaning of the three operands of an object returned by `rectform`.

We start with the expression $x + \sin(y)$, for which `rectform` is able to compute a complete decomposition into real and imaginary part:

```
>> delete x, y: r := rectform(x + sin(y))

 (Re(x) + sin(Re(y)) cosh(Im(y))) +

    I (Im(x) + cos(Re(y)) sinh(Im(y)))
```

The first two operands of `r` are the real and imaginary part of the expression, and the third operand is 0:

```
>> op(r)

 Re(x) + sin(Re(y)) cosh(Im(y)),

    Im(x) + cos(Re(y)) sinh(Im(y)), 0
```

Next we consider the expression $x + f(y)$, where $f(y)$ represents an unknown function in a complex variable. `rectform` can split x into its real and imaginary part, but fails to do this for the subexpression $f(y)$:

```
>> delete f: r := rectform(x + f(y))
```

```
Re(x) + I Im(x) + f(y)
```

The first two operands of the returned object are the real and the imaginary part of x, and the third operand is the remainder $f(y)$, for which `rectform` was not able to extract any information about its real and imaginary part:

```
>> op(r)

Re(x), Im(x), f(y)

>> Re(r), Im(r)

Re(x) + Re(f(y)), Im(x) + Im(f(y))
```

Sometimes `rectform` is not able to extract any information about the real and imaginary part of the input expression. Then the third operand contains the whole input expression, possibly in a rewritten form, due to the recursive mode of operation of `rectform`. The first two operands are 0. Here is an example:

```
>> r := rectform(sin(x + f(y)))

sin(f(y) + I Im(x) + Re(x))

>> op(r)

0, 0, sin(f(y) + I Im(x) + Re(x))

>> Re(r), Im(r)

Re(sin(f(y) + I Im(x) + Re(x))),

   Im(sin(f(y) + I Im(x) + Re(x)))
```

Example 7. Advanced users can extend `rectform` to their own special mathematical functions (see section "Backgrounds" below). To this end, embed your mathematical function into a function environment `f` and implement the behavior of `rectform` for this function as the `"rectform"` slot of the function environment.

If a subexpression of the form `f(u,..)` occurs in `z`, then `rectform` issues the call `f::rectform(u,..)` to the slot routine to determine the rectangular form of `f(u,..)`.

For illustration, we show how this works for the sine function. Of course, the function environment `sin` already has a `"rectform"` slot. We call our function environment `Sin` in order not to overwrite the existing system function `sin`:

```
>> Sin := funcenv(Sin):
   Sin::rectform := proc(u) // compute rectform(Sin(u))
     local r, a, b;
   begin
     // recursively compute rectform of u
     r := rectform(u);

     if op(r, 3) <> 0 then
       // we cannot split Sin(u)
       new(rectform, 0, 0, Sin(u))
     else
       a := op(r, 1); // real part of u
       b := op(r, 2); // imaginary part of u
       new(rectform, Sin(a)*cosh(b), cos(a)*sinh(b), 0)
     end_if
   end:
```

```
>> delete z: rectform(Sin(z))

       Sin(Re(z)) cosh(Im(z)) + (cos(Re(z)) sinh(Im(z))) I
```

If the `if` condition is true, then `rectform` is unable to split `u` completely into its real and imaginary part. In this case, `Sin::rectform` is unable to split `Sin(u)` into its real and imaginary part and indicates this by storing the whole expression `Sin(u)` in the third operand of the resulting `rectform` object:

```
>> delete f: rectform(Sin(f(z)))

                            Sin(f(z))
```

```
>> op(%)

                          0, 0, Sin(f(z))
```

Background:

- If a subexpression of the form `f(u,..)` occurs in `z` and `f` is a function environment, then `rectform` attempts to call the slot `"rectform"` of `f` to determine the rectangular form of `f(u,..)`. In this way, you can extend the functionality of `rectform` to your own special mathematical functions.

  The slot `"rectform"` is called with the arguments `u,..` of `f`. If the slot routine `f::rectform` is not able to determine the rectangular form of `f(u,..)`, then it should return `new(rectform(0,0,f(u,..)))`. See example 7. If `f` does not have a slot `"rectform"`, then `rectform` returns the object `new(rectform(0,0,f(u,..)))` for the corresponding subexpression.

- Similarly, if an element `d` of a library domain `T` occurs as a subexpression of `z`, then `rectform` attempts to call the slot `"rectform"` of that domain with `d` as argument to compute the rectangular form of `d`.

  If the slot routine `T::rectform` is not able to determine the rectangular form of `d`, then it should return `new(rectform(0,0,d))`.

  If the domain `T` does not have a slot `"rectform"`, then `rectform` returns the object `new(rectform(0,0,d))` for the corresponding subexpression.

`register` – remove the memory limit of the demo version

`register(Name, Key)` registers the MuPAD installation on UNIX platforms.

Call(s):

- `register(Name, Key)`

Parameters:

`Name` — the name entry of the registration code: a string
`Key` — the registration key: a string

Return Value: `TRUE` if the registration was successful, and otherwise `FALSE`.

Further Documentation: See the MuPAD license agreement, which can be obtained from `http://www.sciface.com/mupad_download/reg_form.html`.

Details:

- The free MuPAD versions that you can download from the web have a built-in memory limit of 6 megabytes. `register` removes this memory limit on UNIX platforms.

  On Windows platforms, you can register your MuPAD version via the item “Register” of the “Help” menu.

  On Macintosh platforms, choose “About MuPAD” in the Apple menu and then “Register”.

- You obtain a registration key via the following web page:
`http://www.sciface.com/mupad_download/reg_form.html`

- You need write access to the directory tree where MuPAD was installed in order to register. If unsure, register as user *root*. Cf. example 3.

Example 1. If the key is correct and the registration was successful, `register` returns `TRUE`:

```
>> register("My name", "12345-67890-ABCDE")

 Memory limitation removed.

                              TRUE
```

Example 2. If you enter an invalid key, you will get the following message:

```
>> register("My name", "invalid key")

 Wrong password or not registered user.

                              FALSE
```

Example 3. If the key is correct, but you have no write permission to the directory tree where MuPAD was installed, the following happens:

```
>> register("My name", "12345-67890-ABCDE")

 Cannot remove memory limitation.

                              FALSE
```

`reset` – re-initialize a MuPAD session

`reset()` re-initializes a MuPAD session, so that if behaves like a freshly started session afterwards.

Call(s):

- `reset()`

Return Value: the void object `null()` of type `DOM_NULL`.

Related Functions: `delete`, `quit`

Details:

- `reset` initializes a MuPAD session. After a call of `reset()` the current session will behave like a freshly started MuPAD session. `reset` deletes the values of all identifiers and resets the environment variables to their default values. Finally, the initialization files `sysinit.mu` and `userinit.mu` are read again.
- `reset` is permitted only at interactive level. Within a procedure, an error occurs.
- `reset` is a function of the system kernel.

Example 1. `reset` deletes the values of all identifiers and resets environment variables to their default values:

```
>> a := 1: DIGITS := 5: reset(): a, DIGITS

                              a, 10
```

`return` – exit a procedure

`return(x)` terminates the execution of a procedure and returns `x`.

Call(s):

- `return(x)`

Parameters:

`x` — any MuPAD object

Return Value: x.

Related Functions: `DOM_PROC`, `proc`, `->`

Details:

- Usually, MuPAD ends a procedure when all statements of the procedure body were processed. In this case, the return value of the procedure is the result of the last statement that was executed.

  Alternatively, the call `return(x)` inside a procedure leads to immediate exit from the procedure: `x` is evaluated and becomes the return value of the procedure. Execution proceeds after the point where the procedure was invoked.

- `x` may be an expression sequence, i.e., calls such as `return(x1, x2, ...)` are allowed.
- `return()` returns the void object of type `DOM_NULL`.
- Note that `return` is a function, not a keyword. A statement such as `return x;` works in the programming language C, but causes a syntax error in MuPAD.
- If called outside a procedure, `return(x)` just returns `x`.
- `return` is a function of the system kernel.

Example 1. This example shows the implementation of a maximum function (which, in contrast to the system function `max`, accepts only two arguments). If `x` is larger than `y`, the value of `x` is returned and the execution of the procedure `mymax` stops. Otherwise, `return(x)` is not called. Consequently, `y` is the last evaluated object defining the return value:

```
>> mymax := proc(x : Type::Real, y : Type::Real)
   begin
      if x > y then
        return(x)
      end_if;
      y
   end_proc:

>> mymax(3, 2), mymax(4, 5)

                              3, 5

>> delete mymax:
```

Example 2. `return()` returns the void object:

```
>> f := x -> return(): type(f(anything))

                            DOM_NULL

>> delete f:
```

Example 3. If `return` is called on the interactive level, the evaluated arguments are returned:

```
>> x := 1: return(x, y)
```

```
1, y
```

```
>> delete x:
```

`revert` – revert lists or character strings, invert series expansions

`revert` reverses the ordering of the elements in a list and the ordering of characters in a string. For a series expansion, it returns the functional inverse.

Call(s):

- `revert(object)`

Parameters:

`object` — a list, a character string, or a series expansion of type `Series::Puiseux`

Return Value: an object of the same type as the input object, or a symbolic call of type `"revert"`.

Overloadable by: `object`

Related Functions: `series`, `substring`

Details:

- `revert` is a general function to compute inverses with respect to functional composition, or to reverse the order of operands. This type of functionality may be extended to further types of objects via overloading.
- Currently, the MuPAD library provides functionality for strings and lists, where `revert` reverses the order of the elements or characters, respectively. Further, for series expansions, the functional inverse is returned.
- For all other types of MuPAD objects that do not overload `revert`, the symbolic expression `revert(object)` is returned.

Example 1. `revert` operates on lists and character strings:

```
>> revert([1, 2, 3, 4, 5])

                    [5, 4, 3, 2, 1]

>> revert("nuf si DAPuM ni gnimmargorP")

             "Programming in MuPAD is fun"
```

`revert` operates on series:

```
>> revert(series(sin(x), x)) = series(arcsin(x), x)

              3      5                  3      5
             x    3 x       7          x    3 x       7
         x + -- + ---- + O(x ) = x + -- + ---- + O(x )
             6     40                  6     40
```

The functional inverse of the expansion of `exp` around `x = 0` is the expansion of the inverse function `ln` around `x = exp(0) = 1`:

```
>> revert(series(exp(x), x, 3)) = series(ln(x), x = 1, 2)

                   2
            (x - 1)             3
  (x - 1) - -------- + O((x - 1) ) =
               2


                      2
               (x - 1)             3
     (x - 1) - -------- + O((x - 1) )
                  2
```

Example 2. For all other types of objects, a symbolic function call is returned:

```
>> revert(x + y)

                    revert(x + y)
```

The following series expansion is not of type `Series::Puiseux`. Instead, a generalized expansion of type `Series::gseries` is produced. Consequently, `revert` does not compute an inverse:

```
>> revert(series(exp(-x)/(1 + x), x = infinity, 3))

          /     1          1           1         /     1     \ \
   revert| -------- - --------- + --------- + O| --------- | |
         | x exp(x)    2           3           |  4         | |
         \            x  exp(x)   x  exp(x)    \ x  exp(x) / /
```

`rewrite` – rewrite an expression

`rewrite(f, target)` transforms an expression `f` to a mathematically equivalent form, trying to express `f` in terms of the specified target function.

Call(s):

- `rewrite(f, target)`

Parameters:

`f`	—	an arithmetical or boolean expression
`target`	—	the target function to be used in the representation: one of *andor*, `cot`, `coth`, `diff`, `D`, `exp`, `fact`, `gamma`, `heaviside`, `ln`, `piecewise`, `sign`, *sincos*, *sinhcosh*, `tan`, or `tanh`

Return Value: an arithmetical expression.

Overloadable by: `f`

Further Documentation: Chapter "Manipulating Expressions" of the Tutorial.

Related Functions: `collect`, `combine`, `expand`, `factor`, `normal`, `partfrac`, `rationalize`, `rectform`, `simplify`

Details:

- The target indicates the function that is to be used in the desired representation. Symbolic function calls in `f` are replaced by the target function if this is mathematically valid.
- With the target `exp`, all trigonometric and hyperbolic functions are rewritten in terms of `exp`. Further, the inverse functions as well as `arg` are rewritten in terms of `ln`.
- With the target *sincos*, the functions `tan`, `cot`, `exp`, `sinh`, `cosh`, `tanh`, and `coth` are rewritten in terms of `sin` and `cos`.
- With the target *sinhcosh*, the functions `exp`, `tanh`, `coth`, `sin`, `cos`, `tan`, and `cot` are rewritten in terms of `sinh` and `cosh`.
- With the target `diff`, symbolic calls of the differential operator `D` are rewritten in terms of symbolic calls of the function `diff`. E.g., `D(f)(x)` is converted to `diff(f(x), x)`. A univariate expression `D(f)(x)` is rewritten if `x` is an identifier or an indexed identifier. A multivariate expression

`D([n1, n2, ...], f)(x1, x2, ...)` is rewritten if `x1`, `x2` are *distinct* identifiers or indexed identifiers. Trying to rewrite a multivariate call `D(f)(x1, x2, ...)` of the univariate dervative `D(f)` raises an error.

- With the target `D`, symbolic `diff` calls are rewritten in terms of the differential operator `D`. Derivatives of univariate function calls such as `diff(f(x), x)` are rewritten as `D(f)(x)`. Derivatives of multivariate function calls are expressed via `D([n1, n2, ...], f)`. E.g., `diff(f(x, y), x)` is rewritten as `D([1], f)(x, y)`.
- With the target *andor*, the logical operators `xor`, `==>`, and `<=>` are rewritten in terms of `and`, `or`, and `not`.

Example 1. This example demonstrates the use of `rewrite`:

```
>> rewrite(D(D(f))(x), diff)

                        diff(f(x), x, x)

>> diff(f(x, x), x) = rewrite(diff(f(x, x), x), D)

      diff(f(x, x), x) = D([1], f)(x, x) + D([2], f)(x, x)

>> rewrite(fact(n), gamma), rewrite(gamma(n), fact);

                    gamma(n + 1), fact(n - 1)

>> rewrite(sign(x), heaviside), rewrite(heaviside(x), sign);

                                     sign(x)
               2 heaviside(x) - 1, ------- + 1/2
                                        2

>> rewrite(heaviside(x), piecewise)

    piecewise(1 if 0 < x, heaviside(0) if x = 0, 0 if x < 0)
```

Example 2. Trigonometric functions can be rewritten in terms of `exp`, `sin`, `cos` etc.:

```
>> rewrite(tan(x), exp), rewrite(cot(x), sincos),
   rewrite(sin(x), tan)

                                             / x \
                 2                    2 tan| - |
       I exp(I x)  - I  cos(x)             \ 2 /
     - ---------------, ------, -------------
                 2      sin(x)      / x \2
         exp(I x)  + 1           tan| - |  + 1
                                     \ 2 /
```

```
>> rewrite(arcsinh(x), ln)
```

```
                    2     1/2
          ln(x + (x  + 1)   )
```

Changes:

- The new targets `D` and *andor* were introduced.

`RGB` – predefined color names

`RGB::Name` evaluates to a list `[r, g, b]` representing the color '`Name`' by its red, green and blue contributions according to the RGB color model.

`RGB::ColorNames()` provides a list of all predefined color names.

`RGB::ColorNames(subname)` provides a list of all predefined color names that contain `subname`.

Call(s):

- `RGB::Name`
- `RGB::ColorNames()`
- `RGB::ColorNames(subname)`

Parameters:

`Name`	— the name of a color: an identifier
`subname`	— a part of a color name: an identifier

Return Value: `RGB::Name` evaluates to a list `[r, g, b]` of real floating point numbers between `0.0` and `1.0`. `RGB::ColorNames` returns a list of predefined color names.

Related Functions: `plot2d`, `plotfunc2d`, `plot3d`, `plotfunc3d`

Details:

- The RGB values may be used in plot commands.

Example 1. The basic colors of the RGB model are red, green and blue:

```
>> RGB::Red, RGB::Green, RGB::Blue

        [1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]
```

The following call returns all predefined color names containing 'Olive':

```
>> RGB::ColorNames(Olive)

               [OliveDrab, Olive, OliveGreenDark]
```

The RGB values of these colors are:

```
>> RGB::OliveDrab, RGB::Olive, RGB::OliveGreenDark

 [0.419599, 0.556902, 0.137303],

    [0.230003, 0.370006, 0.170003],

    [0.333293, 0.419599, 0.184301]
```

Example 2. The following command plots a filled grey triangle with black border lines on a white background:

```
>> plot2d(BackGround = RGB::White,
          ForeGround = RGB::Black,
          Labeling = TRUE,
          [Mode = List, [polygon(point(0, 0),
                                 point(1, 0),
                                 point(0, 1),
                                 Closed = TRUE,
                                 Filled = TRUE,
                                 Color = RGB::LightGrey)]
          ])
```

---

save – save the state of an identifier

In a procedure, the statement 'save x;' saves the state of the global identifier x.

Call(s):

- save x1, x2, ...
- _save(x1, x2, ...)

Parameters:

`x1, x2, ...` — symbols evaluating to identifiers

Return Value: the void object of type `DOM_NULL`.

Related Functions: `proc`

Details:

- The `save` statement saves the states of identifiers—i.e., their values and properties—during the execution of procedures. The original state of the identifiers is restored when procedure execution is finished. This holds even when an error occurs.

  The `save` statement is to be used only inside the body of a procedure. It cannot be called on the interactive level.

- The arguments of the `save` statement are evaluated as usual. In the statement '`save x;`', the symbol `x` must evaluate to an identifier `y`, say. It is the state of the identifier `y` that is saved.

- The `save` statement is very similar to the `save` declaration for procedures. The main difference to the declaration is that, in order to make the declaration, one has to know the names of the identifiers to be saved in advance. The `save` statement allows to save identifiers which are known only at run-time.

- The `save` statement is usually used in order to temporarily change the properties of an identifier, for example by calling the function `assume`. Eventually, the original properties of the identifiers are restored even if an error occurs.

- The statement '`save x1, x2, ...;`' is equivalent to the function call `_save(x1, x2, ...)`.

- `_save` is a function of the system kernel.

Example 1. First, we define a property for the identifier `y`:

```
>> assume(y < 0)

                                  < 0
```

The properties of the identifier stored in `x` are changed temporarily during the execution of the following procedure `p`:

```
>> p := proc(x : DOM_IDENT)
   begin
       save x;
       assume(x > 0);
       is(x > 0)
   end_proc:
```

From the procedure's result, we see that the properties of `y` were changed during the execution of `p`:

```
>> p(y)

                              TRUE
```

However, the original properties were restored after exiting `p`. The identifier `y` has its original properties:

```
>> is(y > 0), is(y < 0)

                           FALSE, TRUE
```

The restoration of the original properties is guaranteed even if some error occurs inside the procedure. The following procedure `q` raises an error after changing the identifier given by `x`:

```
>> q := proc(x : DOM_IDENT)
   begin
       save x;
       assume(x > 0);
       error("some error")
   end_proc:
   q(y)

 Error: some error [q]
```

Nevertheless, the original assumptions about `y` are restored:

```
>> is(y > 0), is(y < 0)

                           FALSE, TRUE

>> unassume(y): delete p, q:
```

Changes:

- `save` is a new statement.
- In previous MuPAD releases, identifiers could only be saved with the `save` declaration for procedures. Now, the `save` statement may also be used to save an identifier at run-time.

`select` – select operands

`select(object, f)` returns a copy of the object with all operands removed that do not satisfy a criterion defined by the procedure `f`.

Call(s):

- `select(object, f <, p1, p2, ...>)`

Parameters:

`object`	—	a list, a set, a table, an expression sequence, or an expression of type `DOM_EXPR`
`f`	—	a procedure returning a Boolean value
`p1, p2, ...`	—	any MuPAD objects accepted by `f` as additional parameters

Return Value: an object of the same type as the input object.

Overloadable by: `object`

Related Functions: `map`, `op`, `split`, `zip`

Details:

- `select` is a fast and handy function for picking out elements of lists, sets, tables etc. that satisfy a criterion set by the procedure `f`.
- The function `f` must return a value that can be evaluated to one of the Boolean values `TRUE`, `FALSE`, or `UNKNOWN`. It may either return one of these values directly, or it may return an equation or an inequality that can be simplified to one of these values by the function `bool`.
- Internally, the function `f` is applied to all operands `x` of the input object via the call `f(x, p1, p2, ...)`. If the result is not `TRUE`, this operand is removed. The original object is not modified in this process.

  The output object is of the same type as the input object, i.e., a list yields a list, a set yields a set etc.
- An input object that is an expression sequence is not flattened. Cf. example 2.
- Also "atomic" objects such as numbers or identifiers can be passed to `select` as first argument. Such objects are handled like sequences with a single operand.

⊞ `select` is a function of the system kernel.

Example 1. `select` handles lists and sets. In the first example, we select all true statements from a list of logical statements. The result is again a list:

```
>> select([1 = 1, 1 = 2, 2 = 1, 2 = 2], bool)

                    [1 = 1, 2 = 2]
```

In the following example, we extract the subset of all elements that are recognized as zero by `iszero`:

```
>> select({0, 1, x, 0.0, 4*x}, iszero)

                       {0, 0.0}
```

`select` also works on tables:

```
>> T:= table(1 = "y", 2 = "n", 3 = "n", 4 = "y", 5 = "y"):
   select(T, has, "y")

                      table(
                        5 = "y",
                        4 = "y",
                        1 = "y"
                      )
```

The following expression is a sum, i.e., an expression of type `"_plus"`. We extract the sum of all terms that do not contain `x`:

```
>> select(x^5 + 2*x + y - 4, _not@has, x)

                         y - 4
```

We extract all factors containing `x` from the following product. The result is a product with exactly one factor, and therefore, is not of the syntactical type `"_mult"`:

```
>> select(11*x^2*y*(1 - y), has, x)

                           2
                          x
```

```
>> delete T:
```

Example 2. `select` works for expression sequences:

```
>> select((1, -4, 3, 0, -5, -2), testtype, Type::Negative)
```

```
-4, -5, -2
```

The $ command generates such expression sequences:

```
>> select(i $ i = 1..20, isprime)
```

```
2, 3, 5, 7, 11, 13, 17, 19
```

Atomic objects are treated as expression sequences of length one:

```
>> select(5, isprime)
```

```
5
```

The following result is the void object `null()` ob type `DOM_NULL`:

```
>> domtype(select(6, isprime))
```

```
DOM_NULL
```

Example 3. It is possible to pass an "anonymous procedure" to `select`. This allows to perform more complex actions with one call. In the following example, the command `anames(All)` returns a set of all identifiers that have a value in the current MuPAD session. The `select` statement extracts all identifiers beginning with the letter `"h"`:

```
>> select(anames(All), x -> expr2text(x)[0] = "h")
```

```
{has, hold, help, hull, hastype, history, heaviside, hypergeom}
```

`series` – compute a (generalized) series expansion

`series(f, x = x0)` computes the first terms of a series expansion of `f` with respect to the variable `x` around the point `x0`.

Call(s):

- `series(f, x` < `= x0`> <`, order`> <`, dir`> <`, ` *NoWarning*>`)`

Parameters:

`f`	—	an arithmetical expression representing a function in `x`
`x`	—	an identifier
`x0`	—	the expansion point: an arithmetical expression. If not specified, the default expansion point 0 is used.
`order`	—	the number of terms to be computed: a nonnegative integer or `infinity`. The default order is given by the environment variable `ORDER` (default value 6).

Options:

`dir`	—	either *Left*, *Right*, *Real*, or *Undirected*. If no expansion exists that is valid in the complex plane, this argument can be used to request expansions that only need to be valid along the real line. The default is *Undirected*.
NoWarning	—	supresses warning messages printed during the series computation. This can be useful if `series` is called within user-defined procedures.

Return Value: If `order` is a nonnegative integer, then `series` returns either an object of the domain type `Series::Puiseux` or `Series::gseries`, an expression of type `"series"`, or, if `f` is a `RootOf` expression, a set of type `Type::Set`. If `order = infinity`, then `series` returns an arithmetical expression.

Side Effects: The function is sensitive to the environment variable `ORDER`, which determines the default number of terms in series computations.

Overloadable by: `f`

Related Functions: `asympt`, `limit`, `O`, `ORDER`, `RootOf`, `Series::gseries`, `Series::Puiseux`, `solve`, `taylor`, `Type::Series`

Details:

- `series` tries to compute either the Taylor series, the Laurent series, the Puiseux series, or a generalized series expansion of `f` around `x = x0`. See `Series::gseries` for details on generalized series expansions.

  The mathematical type of the series returned by `series` can be queried using the type expression `Type::Series`.

- If `series` cannot compute a series expansion of `f`, a symbolic function call is returned. This is an expression of type `"series"`. Cf. example 11.

- Mathematically, the expansion computed by `series` is valid in some neighborhood of the expansion point in the complex plane. Usually, this is an open disc centered at `x0`. However, if the expansion point is a branch point, then the returned expansion may not approximate the function `f` for values of `x` close to the branch cut. Cf. example 12.

Using the options *`Left`* or *`Right`*, one can compute directed expansions that are valid along the real axis. With the option *`Real`*, a two-sided expansion along the real axis is computed. Cf. examples 5 and 6.

- If `x0` is `infinity` or `-infinity`, then a directed series expansion along the real axis from the left to the positive real infinity or from the right to the negative real infinity, respectively, is computed. If `x0` is `complexInfinity` and `dir` is not specified or *`Undirected`*, then an undirected series expansion around the complex infinity, i.e., the north pole of the Riemann sphere, is computed. Specifying `x0 = infinity` is equivalent to `x0 = complexInfinity` and `dir =` *`Left`*. Similarly, `x0 = -infinity` is equivalent to `x0 = complexInfinity` and `dir =` *`Right`*. Cf. example 7.

  Such a series expansion is computed as follows: The series variable `x` in `f` is replaced by $x = 1/u$ (or $x = -1/u$ for `x0 = -infinity`). Then, a series expansion of `f` around $u = 0$ is computed. Finally, $u = 1/x$ (or $u = -1/x$, respectively) is substituted in the result.

  Mathematically, the result of such a series expansion is a series in $1/x$. However, it may happen that the coefficients of the returned series depend on the series variable. See the corresponding paragraph below.

- The number of requested terms for the expansion is `order` if specified. Otherwise, the value of the environment variable `ORDER` is used. One can change the default value 6 by assigning a new value to `ORDER`.

  The number of terms is counted from the lowest degree term on for finite expansion points, and from the highest degree term on for expansions around infinity, i.e., "`order`" has to be regarded as a "relative truncation order".

  `series` implements a limited amount of precision management to circumvent cancellation. If the number of terms of the computed expansion is less than `order`, a second series computation with a higher value of `order` is tried automatically, and the result of the latter is returned.

  Nevertheless, the actual number of terms in the resulting series expansion may differ from the requested number of terms. Cf. examples 14 and 13.

  NOTE

- In some cases, when cancellation occurs, it may happen that the requested order is too small to compute a series expansion. In such a case, the computation is aborted with an error message. Cf. example 14.

- Expansions of symbolic integrals can be computed. Cf. example 15.

- If `f` is an expression of type `"RootOf"`, then `series` returns the set of all nonzero series solutions of the corresponding algebraic equation. Cf. example 9.

- If order has the value infinity, then the system tries to convert the first argument into a formal infinite series, i.e., it computes a general formula for the n-th coefficient in the Taylor expansion of f. The result is an inactive symbolic sum or a polynomial expression. Cf. example 10.

- If series returns a series expansion of domain type Series::Puiseux, it may happen that the "coefficients" of the returned series depend on the series variable. In this case, the expansion is not a proper Puiseux series in the mathematical sense. Cf. examples 7 and 8. However, if the series variable is x and the expansion point is x_0, then the following is valid for each coefficient function $c(x)$ and every positive ε: $c(x)(x-x_0)^{\varepsilon}$ converges to zero and $c(x)(x-x_0)^{-\varepsilon}$ is unbounded when x approaches x_0. Similarly, if the expansion point is ∞, then, for every positive ε, $c(x)x^{-\varepsilon}$ converges to zero and $c(x)x^{\varepsilon}$ is unbounded when x approaches ∞.

- The function returns a domain object that can be manipulated by the standard arithmetical operations. Moreover, the following methods are available: ldegree returns the exponent of the leading term; Series::Puiseux::order returns the exponent of the error term; expr converts to an arithmetical expression, removing the error term; coeff(s, n) returns the coefficient of the term of s with exponent n; lcoeff returns the leading coefficient; revert computes the inverse with respect to composition; diff and int differentiate and integrate a series expansion, respectively; map applies a function to all coefficients. See the help pages for Series::Puiseux and Series::gseries for further details.

- series works on a symbolic level and should not be called with arguments containing floating point arguments. NOTE

Example 1. We compute a series expansion of $\sin(x)$ around $x = 0$. The result is a Taylor series:

```
>> s := series(sin(x), x)

                 3     5
                x     x        7
            x - -- + --- + O(x )
                6    120
```

Syntactically, the result is an object of domain type Series::Puiseux:

```
>> domtype(s)

                 Series::Puiseux
```

The mathematical type of the series expansion can be queried using the type expression Type::Series:

```
>> testtype(s, Type::Series(Taylor))
```

TRUE

Various system functions are overloaded to operate on series objects. E.g., the function `coeff` can be used to extract the coefficients of a series expansion:

```
>> coeff(s, 5)
```

```
1/120
```

The standard arithmetical operators can be used to add or multiply series expansions:

```
>> s + 2*s, s*s
```

```
             3    5                    4      6
            x    x       7    2   x     2 x       8
      3 x - -- + -- + O(x ), x  - -- + ---- + O(x )
             2   40                   3    45
```

```
>> delete s:
```

Example 2. This example computes the composition of `s` by itself, i.e. the series expansion of $\sin(\sin(x))$.

```
>> s := series(sin(x), x): s @ s = series(sin(sin(x)), x)
```

```
             3    5                3    5
            x    x       7        x    x       7
        x - -- + -- + O(x ) = x - -- + -- + O(x )
             3   10                3   10
```

```
>> delete s:
```

Example 3. We compute the series expansion of the tangent function around the origin in two ways:

```
>> series(sin(x), x) / series(cos(x), x) = series(tan(x), x)
```

```
          3      5                 3      5
         x    2 x       7         x    2 x       7
     x + -- + ---- + O(x ) = x + -- + ---- + O(x )
          3    15                 3    15
```

```
>> bool(%)
```

TRUE

Example 4. We compute a Laurent expansion around the point 1:

```
>> s := series(1/(x^2 - 1), x = 1)

                                   2           3           4
     1             (x - 1)   (x - 1)     (x - 1)     (x - 1)
 --------- - 1/4 + ------- - -------- + -------- - -------- +
 2 (x - 1)            8          16          32          64

             5
    O((x - 1) )

>> testtype(s, Type::Series(Taylor)),
   testtype(s, Type::Series(Laurent))

                              FALSE, TRUE
```

Example 5. Without an optional argument or with the option *Undirected*, the `sign` function is not expanded:

```
>> series(x*sign(x^2 + x), x) =
   series(x*sign(x^2 + x), x, Undirected)

                   2       7                   2       7
         x sign(x + x ) + O(x ) = x sign(x + x ) + O(x )
```

Some simplification occurs if one requests an expansion that is valid along the real axis only:

```
>> series(x*sign(x^2 + x), x, Real)

                                     7
                       x sign(x) + O(x )
```

The `sign` vanishes from the result if one requests a one-sided expansion along the real axis:

```
>> series(x*sign(x^2 + x), x, Right),
   series(x*sign(x^2 + x), x, Left)

                         7             7
                  x + O(x ), - x + O(x )
```

Example 6. In MuPAD, the `heaviside` function is defined only on the real axis. Thus an undirected expansion in the complex plane does not make sense:

```
>> series(x*heaviside(x + 1), x)

 Warning: Could not find undirected series expansion; try option\
 'Left', 'Right', or 'Real' [Series::main]

                   series(x heaviside(x + 1), x)
```

After specifying corresponding options, the system computes an expansion along the real axis:

```
>> series(x*heaviside(x + 1), x, Real),
   series(x*heaviside(x + 1), x, Right)

                           7          7
                   x + O(x ), x + O(x )
```

At the point `I` in the complex plane, the function `heaviside` is not defined, and neither is a series expansion:

```
>> series(heaviside(x), x = I, Real)

 Error: heaviside is not defined for non-real expansion points \
 [heaviside::series]
```

Example 7. We compute series expansions around infinity:

```
>> s1 := series((x + 1)/(x - 1), x = complexInfinity)

                  2    2    2    2    2    /  1 \
              1 + - + -- + -- + -- + -- + O| -- |
                  x    2    3    4    5    |  6 |
                      x    x    x    x     \ x  /

>> s2 := series(psi(x), x = infinity)

                    1      1        1      /  1 \
            ln(x) - --- - ----- + ------ + O| -- |
                    2 x       2        4   |  6 |
                          12 x    120 x    \ x  /

>> domtype(s1), domtype(s2)

                Series::Puiseux, Series::Puiseux
```

Although both expansions are of domain type `Series::Puiseux`, `s2` is not a Puiseux series in the mathematical sense, since the first term contains a logarithm, which has an essential singularity at infinity:

```
>> testtype(s1, Type::Series(Puiseux)),
   testtype(s2, Type::Series(Puiseux))
```

```
                              TRUE, FALSE
```

```
>> coeff(s2)
```

```
                  ln(x), -1/2, -1/12, 0, 1/120
```

The following expansion is of domain type `Series::gseries`:

```
>> s3 := series(exp(x)/(1 - x), x = infinity, 4)
```

```
     exp(x)   exp(x)   exp(x)   exp(x)     / exp(x) \
   - ------ - ------ - ------ - ------ + O| ------ |
       x        2        3        4       |    5   |
               x        x        x        \   x    /
```

```
>> domtype(s3)
```

```
                            Series::gseries
```

```
>> delete s1, s2, s3:
```

Example 8. Oscillating but bounded functions may appear in the "coefficients" of a series expansion as well:

```
>> s := series(sin(x + 1/x), x = infinity)
```

```
           cos(x)   sin(x)   cos(x)   sin(x)   cos(x)    /  1 \
  sin(x) + ------ - ------ - ------ + ------ + ------ + O| -- |
             x         2        3        4        5      |  6 |
                    2 x      6 x      24 x    120 x      \ x  /
```

```
>> domtype(s), testtype(s, Type::Series(Puiseux))
```

```
                      Series::Puiseux, FALSE
```

```
>> coeff(s, -1)
```

```
                              cos(x)
```

Example 9. The algebraic equation $y^5 - y - x = 0$ cannot be resolved in terms of radicals:

```
>> solve(y^5 - y - x, y)

                                  5
                        RootOf(X9  - X9 - x, X9)
```

However, `series` can compute all series solutions of this equation around $x = 0$:

```
>> series(%, x = 0)

 {                                   2      3        4    5
 {       5      7          x    5 x    5 x    385 x    x       6
 { - x - x  + O(x ), - 1 + - + ---- + ---- + ------ + -- + O(x ),
 {                         4    32     32     2048     4

                2      3        4    5
       x    5 x    5 x    385 x    x       6
    1 + - - ---- + ---- - ------ + -- + O(x ),
       4    32     32     2048     4

                               3                     5
        x            2    5 x                  4    x       6
    - I + - - 5/32 I x  - ---- + 385/2048 I x  + -- + O(x ),
        4                 32                        4

                             3                     5         }
       x           2    5 x                  4    x       6  }
    I + - + 5/32 I x  - ---- - 385/2048 I x  + -- + O(x ) }
       4                 32                       4          }
```

It may happen that the series solutions themselves are expressed in terms of `RootOf`'s:

```
>> series(RootOf(y^5 -(x + 2*x^2)*y^3 - x^3*y^2
                 + (x^3 + x^4)*y + x^4 + x^5, y), x)

 {          3/2    5/2    7/2      9/2      11/2
 {    1/2  x      x      x      5 x      7 x            13/2
 { - x    - ---- + ---- - ---- + ------ - ------- + O(x    ),
 {           2      8      16     128      256

            3/2    5/2    7/2      9/2      11/2             }
      1/2  x      x      x      5 x      7 x            13/2  }
     x    + ---- - ---- + ---- - ------ + ------- + O(x    ) }
             2      8      16     128      256               }

     union { x*z1 + O(x^7) |  z1 in RootOf(z1 - z1^3 + 1, z1) }
```

The coefficients of the algebraic equation are allowed to be transcendental. They are internally converted into Puiseux series by `series`:

```
>> series(RootOf(y^3 - y - exp(x - 1) + 1, y), x = 1, 4)

 {                    2           3
 {             (x - 1)    7 (x - 1)            4
 { - (x - 1) - -------- - ---------- + O((x - 1) ),
 {                2           6

                      2           3
      (x - 1)   (x - 1)    5 (x - 1)            4
    1 + ------- - -------- + ---------- + O((x - 1) ),
          2          8          24

                          2             3              }
        (x - 1)   5 (x - 1)    23 (x - 1)            4  }
    - 1 + ------- + ---------- + ----------- + O((x - 1) ) }
            2           8            24                 }
```

An error occurs if some coefficient cannot be expanded into a Puiseux series:

```
>> series(RootOf(y^3 - y - exp(x), y), x = infinity)

 Error: cannot expand coefficients of RootOf(y^3 - y - exp(1/x), y)\
 into Puiseux series [Series::algebraic]
```

Example 10. In this example, we compute a formula for the n-th coefficient a_n in the Taylor expansion of the function $\exp(-x) = \sum_{n\geq 0} a_n x^n$ around zero, by specifying `infinity` as `order`. The result is a symbolic sum:

```
>> series(exp(-x), x, infinity)

                 /   n1     n1                      \
                 |  x   (-1)                        |
              sum| ------------, n1 = 0..infinity |
                 \ n1 gamma(n1)                     /
```

If the input is a polynomial expression, then so is the output:

```
>> series(x^5 - 1, x = 1, infinity)

                          2             3            4          5
  5 (x - 1) + 10 (x - 1)  + 10 (x - 1)  + 5 (x - 1)  + (x - 1)
```

Example 11. No asymptotic expansion is implemented for the Lambert W-function, and `series` returns a symbolic function call:

```
>> series(lambertW(x), x = infinity)

                 series(lambertW(x), x = infinity)

>> domtype(%), type(%)

                        DOM_EXPR, "series"
```

Example 12. The branch cut of the logarithm and the square root is the negative real axis. For a series expansion on the branch cut, `series` uses the function `signIm` to return an expansion that is valid in an open disc around the expansion point:

```
>> series(ln(x), x = -1, 3)

                                      2           3
                               (x + 1)     (x + 1)             4
  I PI signIm(x) - (x + 1) - -------- - -------- + O((x + 1) )
                                  2           3

>> series(sqrt(x), x = -1, 3)

                               signIm(x)
     signIm(x)                 ---------
     ---------                     2
         2        (x + 1) (-1)
  (-1)          - --------------------- -
                              2

                  signIm(x)
                  ---------
           2          2
     (x + 1)  (-1)                      3
     ---------------------- + O((x + 1) )
                 8
```

The situation is more intricate when the expansion point is a branch point. The following expansion of the function `arcsin(x + 1)` is valid in an open disc around the branch point 0:

```
>> series(arcsin(x + 1), x, 4)
```

```
                        1/2     1/2       2  1/2     1/2
PI     1/2      1/2   x 2    (-x)      3 x  2    (-x)
-- - 2    (-x)    + -------------- - ----------------- +
 2                       12                 160

      3  1/2     1/2
   5 x  2    (-x)         9/2
   ----------------- + O(x   )
          896
```

However, the expansion of `f = ln(x + I*x^3)` around the branch point 0 that is returned by `series` does not approximate `f` for values of `x` that are close to the negative real axis:

```
>> f := ln(x + I*x^3);
   g := series(f, x, 4);

                                     3
                          ln(x + I x )

                                  2      4
                      ln(x) + I x  + O(x )

>> DIGITS := 20:
   float(subs([f, expr(g)], x = -0.01 + 0.0000001*I));
   delete DIGITS:

 [- 4.605170178938091416 - 3.1415026535903362385 I,

    - 4.605170183938091368 + 3.1416826535897835718 I]
```

The situation is similar for algebraic branch points:

```
>> f := sqrt(x + I*x^3);
   g := series(f, x, 4);

                                  3 1/2
                          (x + I x )

                    1/2          5/2      9/2
                   x    + 1/2 I x    + O(x   )

>> DIGITS := 20:
   float(subs([f, expr(g)], x = -0.01 + 0.0000001*I));
   delete DIGITS:

 [0.00000449999999871937500725 - 0.1000000002512499991 I,

    - 0.00000449999999906875 + 0.10000000012625 I]

>> delete f, g:
```

Example 13. The first six terms, including zeroes, of the following two series expansions agree:

```
>> series(sin(tan(x)), x, 12);
   series(tan(sin(x)), x, 12);

          3    5       7        9           11
         x    x    55 x    143 x    968167 x         13
     x + -- - -- - ----- - ------ - ---------- + O(x  )
         6    40    1008     3456      39916800

          3    5        7       9          11
         x    x    107 x    73 x    41897 x         13
     x + -- - -- - ------ - ----- + --------- + O(x  )
         6    40    5040    24192    39916800
```

If we want to compute the series expansion of the difference `sin(tan(x)) - tan(sin(x))`, cancellation happens and produces too few terms in the result. `series` detects this automatically and performs a second series computation with increased precision:

```
>> series(sin(tan(x)) - tan(sin(x)), x, 6)

           7       9         11
          x    29 x    1913 x         13
        - -- - ----- - -------- + O(x  )
          30    756      75600
```

It may nevertheless happen that the result has too few terms; cf. example 14.

If rational exponents occur in the series expansion, then it may even happen that the result has more than the number of terms requested by the third argument:

```
>> series(x^2*exp(x) + x*sqrt(sin(x)), x, 3)

                            7/2    4
          3/2    2    3    x      x       5
         x    + x  + x  - ---- + -- + O(x )
                           12     2
```

Example 14. In the following example, the specified order for the expansion is too small to compute the reciprocal, due to cancellation:

```
>> series(exp(x), x, 4)

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

```
>> series(1/(exp(x) - 1 - x - x^2/2 - x^3/6), x, 2)

 Error: order too small [Series::Puiseux::_invert]
```

After increasing the order, an expansion is computed, but possibly with fewer terms:

```
>> series(1/(exp(x) - 1 - x - x^2/2 - x^3/6), x, 3);
   series(1/(exp(x) - 1 - x - x^2/2 - x^3/6), x, 4)

                       24    24      /  1 \
                       -- - ---- + O| -- |
                        4      3     |  2 |
                       x    5 x      \ x  /

                24    24      4       12
                -- - ---- + ----- + ----- + O(1)
                 4      3       2   875 x
                x    5 x    25 x
```

Example 15. `series` and `int` support each other. On the one hand, series expansions can be integrated:

```
>> int(series(1/(2 - x), x), x = 0..1)

                       6
                int(O(x ), x = 0..1) + 1327/1920
```

On the other hand, `series` knows how to handle symbolic integrals:

```
>> int(x^x, x)

                                 x
                           int(x , x)

>> series(%, x = 0, 3)

                            /                2        \
      2 / ln(x)       \    3 |   ln(x)   ln(x)          |      4
 x + x  | ----- - 1/4 | + x  | - ----- + ------ + 1/27 | + O(x )
        \   2         /      \     9       6           /

>> int(exp(-x*sin(t)), t = 0.. x)

                  int(exp(-x sin(t)), t = 0..x)

>> series(%, x = 0)
```

```
                3      5
               x    5 x       7
           x - -- + ---- + O(x )
                2    24
```

```
>> int(cos((x*t^2 + x^2*t))^(1/3), t = 0..2)
```

```
                        2    2   1/3
           int(cos(t x  + t  x)    , t = 0..2)
```

```
>> series(%, x)
```

```
                    2      3         4       5
                16 x    4 x    100 x    16 x        6
            2 - ----- - ---- - ------ - ----- + O(x )
                 15      3       81       9
```

Example 16. Users can extend the power of `series` by implementing `series` attributes (slots) for their own special mathematical functions.

We illustrate how to write such a series attribute, using the case of the exponential function. (Of course, this function already has a `series` attribute in MuPAD, which you can inspect via `expose(exp::series)`.) In order not to overwrite the already existing attribute, we work on a copy of the exponential function called `Exp`.

The `series` attribute must be a procedure with four arguments. This procedure is called whenever a series expansion of `Exp` with an arbitrary argument is to be computed. The first argument is the argument of `Exp` in the `series` call. The second argument is the series variable; the expansion point is always the origin 0; other expansion points are internally moved to the origin by a change of variables. The third and the fourth argument are identical with the `order` and the `dir` argument of `series`, respectively.

For example, the command `series(Exp(x^2 + 2), x, 5)` is internally converted into the call `Exp::series(x^2 + x, x, 5, Undirected)`. Here is an example of a `series` attribute for `Exp`.

```
>> // The series attribute for Exp. It handles the call
   // series(Exp(f), x = 0, order, dir)
   ExpSeries := proc(f, x, order, dir)
     local t, x0, s, r, i;
   begin
     // Expand the argument into a series.
     t := series(f, x, order, dir);

     // Determine the order k of the lowest term in t, so that
     // t = c*x^k + higher order terms, for some nonzero c.
     k := ldegree(t);
```

```
    if k = FAIL then
      // t consists only of an error term O(..)
      error("order too small");

    elif k < 0 then
      // This corresponds to an expansion of exp around infinity,
      // which does not exist for the exponential
      // function, since it has an essential singularity. Thus we
      // return FAIL, which makes series return unevaluatedly. For
      // other special functions, you may add an asymptotic
      // expansion here.
      return(FAIL);

    else // k >= 0
      // This corresponds to an expansion of exp around a
      // finite point x0. We write t = x0 + y, where all
      // terms in y have positive order, use the
      // formula exp(x0 + y) = exp(x0)*exp(y) and compute
      // the series expansion of exp(y) as the functional
      // composition of the Taylor series of exp(x) around
      // x = 0 with t - x0. If your special function has
      // any finite singularities, then they should be
      // treated here.
      x0 := coeff(t, x, 0);
      s := Series::Puiseux::create(1, 0, order,
             [1/i! $ i = 0..(order - 1)], x, 0, dir);
      return(Series::Puiseux::scalmult(s @ (t - x0), Exp(x0), 0))
    end_if
  end_proc:
```

This special function must be embedded in a function environment. The following command defines `Exp` as a function environment and copies the code for evaluating the system function `exp`. The `subsop` command achieves that `Exp` with symbolic arguments is returned as `Exp` and not as `exp`, see the help page for `DOM_PROC`.

```
>> Exp := funcenv(subsop(op(exp, 1), 6 = hold(Exp)), NIL, NIL):
   Exp(1), Exp(-1.0), Exp(x^2 + x)

                                                   2
                Exp(1), 0.3678794412, Exp(x + x )
```

`series` can already handle this "new" function, but it can only compute a Taylor expansion with symbolic derivatives:

```
>> ORDER := 3: series(Exp(x), x = 0)
```

```
                    2
                   x  D(D(Exp))(0)      3
1 + x D(Exp)(0) + --------------- + O(x )
                         2
```

One can define the series attribute of Exp by assigning the procedure above to its series slot:

```
>> Exp::series := ExpSeries:
```

Now we can test the new attribute:

```
>> series(Exp(x^2 + x), x = 0) = series(exp(x^2 + x), x = 0)
```

```
                 2                       2
              3 x       3             3 x       3
     1 + x + ---- + O(x ) = 1 + x + ---- + O(x )
               2                       2
```

```
>> series(Exp(x^2 + x), x = 2) = series(exp(x^2 + x), x = 2)
```

```
                                                 2
                             27 Exp(6) (x - 2)              3
 Exp(6) + 5 Exp(6) (x - 2) + ----------------- + O((x - 2) ) =
                                     2


                                                 2
                                27 exp(6) (x - 2)              3
    exp(6) + 5 exp(6) (x - 2) + ----------------- + O((x - 2) )
                                        2
```

```
>> series(Exp(x^2 + x), x = 0, 0)
```

```
 Error: order too small [ExpSeries]
```

```
>> series(Exp(x^2 + x), x = infinity)
```

```
                              2
           series(Exp(x + x ), x = infinity)
```

Another possibility to obtain series expansions of user-defined functions is to define the diff attribute of the corresponding function environment. This is used by series to compute a Taylor expansion when no series attribute exists. However, this only works when a Taylor expansion exists, whilst a series attribute can handle more general types of series expansions as well.

```
>> delete ExpSeries, Exp:
```

Changes:

- Branch points and branch cuts are handled better; cf. example 12.
- The precision management was improved; cf. example 13.
- The new option *Undirected* was introduced.
- Undirected expansions around `complexInfinity` are now possible.
- `series` can now compute series solutions of an algebraic equation specified by a `RootOf` expression. Then the return value is a set of type `Type::Set`. Cf. example 9.
- The internal structure of the datatype `Series::Puiseux`, which is returned by most calls to `series`, has changed. See the corresponding help page for more details.

`setuserinfo` – set an information level

`setuserinfo(f, n)` sets the information level for the function `f` to `n`, thus activating or deactivating `userinfo` commands built into `f`.

Call(s):

- `setuserinfo(f, n <, style>)`
- `setuserinfo(f)`
- `setuserinfo(n)`
- `setuserinfo(NIL)`
- `setuserinfo()`

Parameters:

`f`	—	a procedure, the name of a domain or the flag *Any*
`n`	—	the "information level": a nonnegative integer
`style`	—	either *Name* or *Quiet*

Options:

Name	—	causes `userinfo` to append the name of the calling procedure to the printed message
Quiet	—	causes `userinfo` to suppress the prefix "`Info:`" at the beginning of a line

Return Value: the previously set information level.

Related Functions: `print`, `userinfo`, `warning`

Details:

- The information level controls the printing of information by the function `userinfo`. This function is built into various library routines to display progress information during the execution of algorithms.
- `setuserinfo(f, n <, style>)` sets the information level of `f` to the value `n` and returns the previously set value. Setting an information level for a domain does not change previously set information levels of the methods of this domain.
- `setuserinfo(f)` returns the current information level of `f` without changing it.
- `setuserinfo(`*`Any`*`, n <, style>)` sets the global information level to the value `n` and returns the previously set value. Note, that this does not change previously set information levels of domains and procedures.
- `setuserinfo(n)` is equivalent to `setuserinfo(`*`Any`*`, n)`.
- `setuserinfo(`*`Any`*`)` returns the global information level without changing it.
- `setuserinfo(NIL)` resets the information level of *all* functions and domains to the default value `0`. With this value, no information is printed by `userinfo`.
- `setuserinfo()` returns a table of all previously set information levels. This table is cleared by the call `setuserinfo(NIL)`.

Example 1. We define a procedure `f` that prints information via `userinfo`:

```
>> f := proc(x)
        begin
          userinfo(1, "enter 'f'");
          userinfo(2, "the argument is " . expr2text(x));
          x^2
        end_proc:
```

After activating the `userinfo` commands inside `f` via `setuserinfo`, any call to `f` prints status information:

```
>> setuserinfo(f, 1, Name): f(5)

 Info: enter 'f' [f]

                                25
```

The information level of `f` is increased:

```
>> setuserinfo(f, 2): f(4)

 Info: enter 'f'
 Info: the argument is 4

                                         16
```

The prefix "`Info:`" shall not be printed:

```
>> setuserinfo(f, 2, Quiet): f(3)

 enter 'f'
 the argument is 3

                                         9
```

The `userinfo` commands are deactivated by clearing all information levels globally:

```
>> setuserinfo(NIL): f(2)

                                         4

>> delete f:
```

`share` – create a unique data representation

`share()` creates a unique data representation for every MuPAD object. This function serves a highly technical purpose. Usually, there should be no need for a user to call this function.

Call(s):

- `share()`

Return Value: the void object of type `DOM_NULL`.

Related Functions: `bytes`

Details:

- If `share` is executed, a unique data representation is created for every MuPAD object before the next command is executed on the interactive level. This means that every MuPAD object is only located once in the physical memory. Thus, `share` reduces the number of logical bytes used in a MuPAD session.

- `share` is a very time consuming function which also needs a lot of memory during its execution.
- `share` is not executed immediately; it is only executed on returning to the interactive level. Therefore, it cannot be used in procedures to release memory during a longer computation.
- `share` is a function of the system kernel.

Example 1. The following example was carried out in a fresh MuPAD session. One sees that `share` reduces the number of logical bytes. However, one observes that the kernel needs some extra physical memory for executing the `share` call. The output of the example will differ on different machines:

```
>> int(x, x): bytes()

                        1980600, 2191872, 2147483647

>> share(): bytes()

                        1201076, 2830848, 2147483647
```

`sign` – the sign of a real or complex number

`sign(z)` returns the sign of the number `z`.

Call(s):

- `sign(z)`

Parameters:

`z` — an arithmetical expression

Return Value: an arithmetical expression.

Overloadable by: `z`

Side Effects: `sign` respects properties of identifiers. For real expressions, the result may depend on the value of the environment variable `DIGITS`.

Related Functions: `abs`, `conjugate`, `Im`, `Re`

Details:

- Mathematically, the sign of a complex number $z \neq 0$ is defined as $z/|z|$. For real numbers, this reduces to 1 or -1.
- `sign(0)` and `sign(0.0)` return `0`. The user may redefine this value by a direct assignment, e.g.:

  `unprotect(sign): sign(0) := 1: protect(sign):`
- If the type of `z` is `DOM_INT`, `DOM_RAT`, or `DOM_FLOAT`, a fast kernel function is used to determine the sign. The return value is either -1, 0, or 1.
- If the sign of the expression cannot be determined, a symbolic function call is returned. Certain simplifications are implemented. In particular, numerical factors of symbolic products are simplified. Cf. example 2.
- The `expand` function rewrites the sign of a product to a product of signs. E.g., `expand(sign(x*y))` yields `sign(x)*sign(y)`. Cf. example 2.
- For constant expressions such as `PI - sqrt(2)`, `exp(I*3) - I*sin(3)` etc., internal floating point evaluation is used to determine, whether the expression represents a nonzero real number. If so, the sign -1 or 1 is returned. Internally, the floating point approximation is checked for reliability. Cf. example 4.

Example 1. We compute the sign of various real numbers and expressions:

```
>> sign(-8/3), sign(3.2), sign(exp(3) - sqrt(2)*PI), sign(0)

                          -1, 1, 1, 0
```

The sign of a complex number `z` is the complex number `z/abs(z)`:

```
>> sign(0.5 + 1.1*I), sign(2 + 3*I), sign(exp(sin(2 + 3*I)))

                                                         1/2
 0.4138029443 + 0.9103664775 I, (2/13 + 3/13 I) 13   ,

    exp(I cos(2) sinh(3))
```

Example 2. `sign` yields a symbolic, yet simplified, function call if identifiers are involved:

```
>> sign(x), sign(2*x*y), sign(2*x + y),  sign(PI*exp(2 + y))

       sign(x), sign(x y), sign(2 x + y), sign(exp(y + 2))
```

In special cases, the `expand` function may provide further simplifications:

```
>> expand(sign(2*x*y)), expand(sign(PI*exp(2 + y)))

                   sign(x) sign(y), sign(exp(y))
```

Example 3. `sign` respects properties of identifiers:

```
>> sign(x + PI)

                            sign(x + PI)

>> assume(x > -3): sign(x + PI)

                                  1

>> unassume(x):
```

Example 4. The following rational number approximates π to about 20 digits:

```
>> p := 157079632679489661923/50000000000000000000:
```

With the standard precision `DIGITS` = 10, the float test inside `sign` does not give a decisive answer, whether `p` is larger or smaller than π:

```
>> float(PI - p)

                                 0.0
```

This result is subject to numerical roundoff and does not allow a conclusion on the sign of the number `PI - p`. The float test inside `sign` checks the reliablity of floating point approximations. In this case, no simplified result is returned:

```
>> sign(PI - p)

        sign(PI - 157079632679489661923/50000000000000000000)
```

With increased `DIGITS`, a reliable decision can be taken:

```
>> DIGITS := 20: sign(PI - p)

                                  1

>> delete p, DIGITS:
```

`signIm` – the sign of the imaginary part of a complex number

`signIm(z)` represents the sign of `Im(z)`.

Call(s):

- `signIm(z)`

Parameters:

`z` — an arithmetical expression representing a complex number

Return Value: either ±1, 0, or a symbolic call of type `"signIm"`.

Side Effects: Properties of identifiers set via `assume` are taken into account.

Overloadable by: `z`

Details:

- `signIm(z)` indicates whether the complex number `z` lies in the upper or in the lower half plane: `signIm(z)` yields 1 if `Im(z)`> 0, or if `z` is real and `z`< 0. At the origin: `signIm(0)=0`. For all other numerical arguments, -1 is returned. Thus, `signIm(z)=sign(Im(z))` if `z` is not on the real axis.
- If the position of the argument in the complex plane cannot be determined, then a symbolic call is returned. If appropriate, the reflection rule `signIm(-x)` = `-signIm(x)` is used.
- The functions `diff` and `series` treat `signIm` as a constant function. Cf. example 2.
- The following relation holds for arbitrary complex z and p:

$$(-z)^p = z^p\,(-1)^{-p\,\texttt{signIm}(z)}.$$

Example 1. For numerical values, the position in the complex plane can always be determined:

```
>> signIm(2 + I), signIm(- 4 - I*PI), signIm(0.3), signIm(-2/7),
   signIm(-sqrt(2) + 3*I*PI)

                        1, -1, -1, 1, 1
```

Symbolic arguments without properties lead to symbolic calls:

```
>> signIm(x), signIm(x - I*sqrt(2))

                                         1/2
                   signIm(x), signIm(x - I 2   )
```

Properties set via `assume` are taken into account:

```
>> assume(x, Type::Real): signIm(x - I*sqrt(2))

                                    -1

>> assume(x > 0): signIm(x)

                                    -1

>> assume(x < 0): signIm(x)

                                     1

>> assume(x = 0): signIm(x)

                                     0

>> unassume(x):
```

Example 2. `signIm` is a constant function, apart from the jump discontinuities along the real axis. These discontinuities are ignored by `diff`:

```
>> diff(signIm(z), z)

                                     0
```

Also `series` treats `signIm` as a constant function:

```
>> series(signIm(z/(1 - z)), z = 0)

                    /    z    \      6
              signIm| ------- | + O(z )
                    \ - z + 1 /
```

Changes:

- The reflection `signIm(-x)` = `-signIm(x)` was implemented.

`simplify` – simplify an expression

`simplify(f)` tries to simplify the expression `f` by applying term rewriting rules.

`simplify(f, target)` restricts the simplification to term rewriting rules applicable to the target function(s).

Call(s):

- `simplify(f <, target>)`
- `simplify(l <, target>)`

Parameters:

`f` — an arithmetical expression
`l` — a set, a list, an array, or a polynomial of type `DOM_POLY`

Options:

`target` — one of the identifiers `cos`, `sin`, `exp`, `ln`, `sqrt`, *`logic`*, or *`relation`*

Return Value: an object of the same type as the input object `f` or `l`, respectively.

Overloadable by: `f, l`

Side Effects: Without a target option, `simplify` reacts to properties of identifiers.

Further Documentation: Chapter "Manipulating Expressions" of the Tutorial.

Related Functions: `collect`, `combine`, `expand`, `factor`, `match`, `normal`, `radsimp`, `rectform`, `rewrite`

Details:

- In a call without a target option, first a simplification of the expression "as a whole" is tried. This includes rewriting of products of trigonometric and exponential terms. Next, `simplify` is recursively applied to the operands of the expression. In this process, the `"simplify"` methods of the special functions contained in the expression are called.
- The call `simplify(f)` implies all simplifications that can be achieved with the targets `sin`, `cos`, `exp`, and `ln`.
- The call `simplify(f, sqrt)` is equivalent to `radsimp(f)`. With this option, constant radical expressions are simplified.
- In the call `simplify(l <, target>)`, simplification is applied to the operands of the object `l`.

Option <target>:

- With the targets `sin`, `cos`, `exp`, and `ln`, only specific simplifications such as rewriting of products of trigonometric or exponential terms occur.
- The option `sqrt` invokes `radsimp`, i.e., simplification of constant radicals occur. See `radsimp` for details.
- With the option *`logic`*, rules of Boolean algebra are applied to Boolean expressions; the property mechanism it not used to decide the truth of the atoms.
- The option *`relation`* is obsolete now, since arithmetical operations for equations and inequalities are now available in MuPAD. It is still available for compatibility reasons, but will be removed in future versions.

Example 1. `simplify` tries to simplify algebraic expressions:

```
>> simplify(exp(x)-exp(x/2)^2)

                                 0

>> f := sin(x)^2 + cos(x)^2 + (exp(x) - 1)/(exp(x/2) + 1):
   simplify(f)

                              / x \
                           exp| - |
                              \ 2 /
```

Only special simplifications occur if special target functions are specified:

```
>> simplify(f, sin)

                                    / x \
                       exp(x) + exp| - |
                                    \ 2 /
                       -----------------
                             / x \
                          exp| - | + 1
                             \ 2 /

>> simplify(f, exp)

                   2          2       / x \
             cos(x)  + sin(x)  + exp| - | - 1
                                      \ 2 /

>> delete f:
```

Example 2. The option `sqrt` serves for simplifying radicals:

```
>> simplify(sqrt(4 + 2*sqrt(3)), sqrt)

                              1/2
                             3    + 1

>> x := 1/2 + sqrt(23/108):
   y := x^(1/3) + 1/3/x^(1/3):
   z := y^3 - y

 /                             /  1/2   1/2       \1/3 \3
 |            1                | 3    23          |    |
 | ------------------------- + | ---------- + 1/2 |    |  -
 |   /  1/2   1/2       \1/3   \      18          /    |
 |   | 3    23          |                              |
 | 3 | ---------- + 1/2 |                              |
 \   \      18          /                              /

    /  1/2   1/2       \1/3
    | 3    23          |                 1
    | ---------- + 1/2 |    - -------------------------
    \      18          /        /  1/2   1/2       \1/3
                                | 3    23          |
                              3 | ---------- + 1/2 |
                                \      18          /

>> simplify(z, sqrt)

                                 1

>> delete x, y, z:
```

Example 3. The option *logic* serves for simplifying Boolean expressions:

```
>> simplify((a and b) or (a and (not b)), logic)

                                 a
```

Example 4. User–defined functions can have `"simplify"` attributes. For example, suppose we know that f is an additive function (but we do not know more about f). Hence we cannot compute the function value of f at any point except zero, but we can tell MuPAD to use the additivity:

```
>> f := funcenv( x -> if iszero(x) then 0 else procname(x) end):
   f::simplify := proc(F)
                    local argument;
                    begin
                      argument := op(F,1);
                      if type(argument) = "_plus" then
                        map(argument, f)
                      else
                        F
                      end
                    end:
```

```
>> f(x + 3*y) - f(3*y) = simplify(f(x + 3*y) - f(3*y))

                   f(x + 3 y) - f(3 y) = f(x)
```

We could still refine the `"simplify"` attribute of `f` such that it also turns `f(3*y)` into `3*f(y)`. However, it is certainly a matter of taste whether $f(x) + f(y)$ is really simpler than $f(x+y)$. The reverse rule (rewriting $f(x)+f(y)$ as $f(x+y)$) is not context-free and cannot be implemented in a `"simplify"` attribute.

`sin`, `cos`, `tan`, `csc`, `sec`, `cot` – the trigonometric functions

`sin(x)` represents the sine function.

`cos(x)` represents the cosine function.

`tan(x)` represents the tangent function `sin(x)/cos(x)`.

`csc(x)` represents the cosecant function `1/sin(x)`.

`sec(x)` represents the secant function `1/cos(x)`.

`cot(x)` represents the cotangent function `cos(x)/sin(x)`.

Call(s):

- `sin(x)`
- `cos(x)`
- `tan(x)`
- `csc(x)`
- `sec(x)`
- `cot(x)`

Parameters:

`x` — an arithmetical expression or a floating point interval

Return Value: an arithmetical expression or a floating point interval

Overloadable by: `x`

Side Effects: When called with a floating point argument, the functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Related Functions: `arcsin`, `arccos`, `arctan`, `arccsc`, `arcsec`, `arccot`

Details:

- The arguments have to be specified in radians, not in degrees. E.g., use π to specify an angle of 180^o.
- All trigonometric functions are defined for complex arguments.
- Floating point values are returned for floating point arguments. Floating point intervals are returned for floating point interval arguments. Unevaluated function calls are returned for most exact arguments.
- Translations by integer multiples of π are eliminated from the argument. Further, arguments that are rational multiples of π lead to simplified results; symmetry relations are used to rewrite the result using an argument from the standard interval $[0, \pi/2)$. Explicit expressions are returned for the following arguments:

  $$0,\ \frac{\pi}{2},\ \frac{\pi}{3},\ \frac{\pi}{4},\ \frac{\pi}{5},\ \frac{2\,\pi}{5},\ \frac{\pi}{6},\ \frac{\pi}{8},\ \frac{3\,\pi}{8},\ \frac{\pi}{10},\ \frac{3\,\pi}{10},\ \frac{\pi}{12},\ \frac{5\,\pi}{12}\ .$$

  Cf. example 2.
- The result is rewritten in terms of hyperbolic functions, if the argument is a rational multiple of `I`. Cf. example 3.
- The functions `expand` and `combine` implement the addition theorems for the trigonometric functions. Cf. example 4.
- The trigonometric functions do not respond to properties set via `assume`. Use `simplify` to take such properties into account. Cf. example 4.
- `sec(x)` and `csc(x)` are immediately rewritten as `1/cos(x)` and `1/sin(x)`, respectively. Use `expand` or `rewrite` to rewrite expressions involving `tan` and `cot` in terms of `sin` and `cos`. Cf. example 5.
- The inverse functions are implemented by `arcsin`, `arccos`, `arctan`, `arccsc`, `arcsec`, and `arccot`, respectively. Cf. example 6.
- The float attributes are kernel functions, i.e., floating point evaluation is fast.

Example 1. We demonstrate some calls with exact and symbolic input data:

```
>> sin(PI), cos(1), tan(5 + I), csc(PI/2), sec(PI/11), cot(PI/8)

                                               1       1/2
          0, cos(1), tan(5 + I), 1, ---------, 2    + 1
                                       / PI \
                                    cos| -- |
                                       \ 11 /
```

```
>> sin(-x), cos(x + PI), tan(x^2 - 4)

                                         2
                  -sin(x), -cos(x), tan(x  - 4)
```

Floating point values are computed for floating point arguments:

```
>> sin(123.4), cos(5.6 + 7.8*I), cot(1.0/10^20)

       -0.7693905459, 946.4239673 + 770.3351731 I, 1.0e20
```

Floating point intervals are computed for interval arguments:

```
>> sin(0 ... 1), cos(20 ... 30), tan(0 ... 5)

     0.0 ... 0.8414709849, -1.0 ... 1.0, RD_NINF ... RD_INF
```

For the functions with discontinuities, the result may be a union of intervals:

```
>> csc(-1 ... 1), tan(1 ... 2)

 RD_NINF ... -1.188395105 union 1.188395105 ... RD_INF,

    RD_NINF ... -2.022053061 union 1.557407724 ... RD_INF
```

Example 2. Some special values are implemented:

```
>> sin(PI/10), cos(2*PI/5), tan(123/8*PI), cot(-PI/12)

           1/2         1/2
          5           5            1/2         1/2
          ---- - 1/4, ---- - 1/4, 2    + 1, - 3    - 2
            4           4
```

Translations by integer multiples of π are eliminated from the argument:

```
>> sin(x + 10*PI), cos(3 - PI), tan(x + PI), cot(2 - 10^100*PI)

                 sin(x), -cos(3), tan(x), cot(2)
```

All arguments that are rational multiples of π are transformed to arguments from the interval $[0, \pi/2)$:

```
>> sin(4/7*PI), cos(-20*PI/9), tan(123/11*PI), cot(-PI/13)

          / 3 PI \      / 2 PI \      / 2 PI \        / PI \
       sin| ---- |, cos| ---- |, tan| ---- |, - cot| -- |
          \   7  /      \   9  /      \  11  /        \ 13 /
```

Example 3. Arguments that are rational multiples of I are rewritten in terms of hyperbolic functions:

```
>> sin(5*I), cos(5/4*I), tan(-3*I)

                I sinh(5), cosh(5/4), -I tanh(3)
```

For other complex arguments, use expand to rewrite the result:

```
>> sin(5*I + 2*PI/3), cos(5/4*I - PI/4), tan(-3*I + PI/2)

         / 2 PI         \      /          PI \      / PI        \
      sin| ---- + 5 I |, cos| 5/4 I - -- |, tan| -- - 3 I |
         \   3          /      \           4 /      \  2        /

>> expand(sin(5*I + 2*PI/3)), expand(cos(5/4*I - PI/4)),
   expand(tan(-3*I + PI/2))

  1/2                            1/2
 3    cosh(5)                   2    cosh(5/4)
 ------------ - 1/2 I sinh(5), -------------- +
      2                               2

          1/2                I cosh(3)
   1/2 I 2    sinh(5/4), - ---------
                              sinh(3)
```

Example 4. The expand function implements the addition theorems:

```
>> expand(sin(x + PI/2)), expand(cos(x + y))

              cos(x), cos(x) cos(y) - sin(x) sin(y)
```

The combine function uses these theorems in the other direction, trying to rewrite products of trigonometric functions:

```
>> combine(sin(x)*sin(y), sincos)
```

```
cos(x - y)   cos(x + y)
---------- - ----------
     2            2
```

The trigonometric functions do not immediately respond to properties set via `assume`:

```
>> assume(n, Type::Integer): sin(n*PI), cos(n*PI)

                      sin(n PI), cos(n PI)
```

Use `simplify` to take such properties into account:

```
>> simplify(sin(n*PI)), simplify(cos(n*PI))

                                  n
                           0, (-1)

>> assume(n, Type::Odd): sin(n*PI + x), simplify(sin(n*PI + x))

                     sin(x + n PI), -sin(x)

>> y := cos(x - n*PI) + cos(n*PI - x): y , simplify(y)

            cos(x - n PI) + cos(n PI - x), -2 cos(x)

>> delete n, y:
```

Example 5. Various relations exist between the trigonometric functions:

```
>> csc(x), sec(x)

                           1       1
                        ------, ------
                        sin(x)  cos(x)
```

The function `expand` rewrites all trigonometric functions in terms of `sin` and `cos`:

```
>> expand(tan(x)), expand(cot(x))

                        sin(x)  cos(x)
                        ------, ------
                        cos(x)  sin(x)
```

Use `rewrite` to obtain a representation in terms of a specific target function:

```
>> rewrite(tan(x)*exp(2*I*x), sincos), rewrite(sin(x), cot)
```

```
                                         / x \
                                   2 cot| - |
sin(x) (cos(2 x) + I sin(2 x))          \ 2 /
------------------------------, -------------
            cos(x)                 / x \2
                                cot| - |  + 1
                                   \ 2 /
```

Example 6. The inverse functions are implemented by `arcsin`, `arccos` etc.:

```
>> sin(arcsin(x)), sin(arccos(x)), cos(arctan(x))

                         2 1/2       1
                x, (1 - x )   , -----------
                                  2     1/2
                                (x  + 1)
```

Note that `arcsin(sin(x))` does not necessarily yield `x`, because `arcsin` produces values with real parts in the interval $[-\pi/2, \pi/2]$:

```
>> arcsin(sin(3)), arcsin(sin(1.6 + I))

                   PI - 3, 1.541592654 - 1.0 I
```

Example 7. Various system functions such as `diff`, `float`, `limit`, or `series` handle expressions involving the trigonometric functions:

```
>> diff(sin(x^2), x), float(sin(3)*cot(5 + I))

                   2
          2 x cos(x ), - 0.01668502608 - 0.1112351327 I

>> limit(x*sin(x)/tan(x^2), x = 0)

                                1

>> series((tan(sin(x)) - sin(tan(x)))/sin(x^7), x = 0)

                         2          4
                     29 x    1913 x       6
              1/30 + ----- + ------- + O(x )
                      756     75600
```

Changes:

- floating point intervals are handled
- `diff(cot(x), x)` now returns `-cot(x)^2 - 1` instead of `-1/sin(x)^2`.

`sinh, cosh, tanh, csch, sech, coth` – the hyperbolic functions

`sinh(x)` represents the hyperbolic sine function.

`cosh(x)` represents the hyperbolic cosine function.

`tanh(x)` represents the hyperbolic tangent function `sinh(x)/cosh(x)`.

`csch(x)` represents the hyperbolic cosecant function `1/sinh(x)`.

`sech(x)` represents the hyperbolic secant function `1/cosh(x)`.

`coth(x)` represents the hyperbolic cotangent function `cosh(x)/sinh(x)`.

Call(s):

- `sinh(x)`
- `cosh(x)`
- `tanh(x)`
- `csch(x)`
- `sech(x)`
- `coth(x)`

Parameters:

`x` — an arithmetical expression or a floating point interval

Return Value: an arithmetical expression or a floating point interval

Overloadable by: `x`

Side Effects: When called with a floating point argument, the functions are sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Related Functions: `arcsinh`, `arccosh`, `arctanh`, `arccsch`, `arcsech`, `arccoth`

Details:

- Theses functions are defined for complex arguments.
- Floating point values are returned for floating point arguments. Floating point intervals are returned for floating point interval arguments. Unevaluated function calls are returned for most exact arguments.
- Arguments that are integer multiples of $i\,\pi/2$ lead to simplified results. If the argument involves a negative numerical factor of `Type::Real`, then symmetry relations are used to make this factor positive. Cf. example 2.
- The special values

  sinh(0) = 0, sinh(±`infinity`) = ±`infinity`,

  cosh(0) = 1, cosh(±`infinity`) = `infinity`,

  tanh(0) = 0, tanh(±`infinity`) = ±1,

  coth(±`infinity`) = ±1

  are implemented.
- The functions `expand` and `combine` implement the addition theorems for the hyperbolic functions. Cf. example 3.
- `sech(x)` and `csch(x)` are rewritten as `1/cosh(x)` and `1/sinh(x)`, respectively. Use `expand` or `rewrite` to rewrite expressions involving `tanh` and `coth` in terms of `sinh` and `cosh`. Cf. example 4.
- The inverse functions are implemented by `arcsin`, `arccos`, `arctan`, `arccsc`, `arcsec`, and `arccot`, respectively. Cf. example 5.
- The float attributes are kernel functions, i.e., floating point evaluation is fast.

Example 1. We demonstrate some calls with exact and symbolic input data:

```
>> sinh(I*PI), cosh(1), tanh(5 + I), csch(PI), sech(1/11), coth(8)

                                      1          1
     0, cosh(1), tanh(5 + I), --------, ----------, coth(8)
                              sinh(PI)  cosh(1/11)
```

```
>> sinh(x), cosh(x + I*PI), tanh(x^2 - 4)

                                                 2
              sinh(x), cosh(x + I PI), tanh(x  - 4)
```

Floating point values are computed for floating point arguments:

```
>> sinh(123.4), cosh(5.6 + 7.8*I), coth(1.0/10^20)
```

```
        1.953930316e53, 7.295585032 + 135.0143985 I, 1.0e20
```

For floating point intervals, intervals enclosing the image are calculated:

```
>> cosh(-1 ... 1), tanh(-1 ... 1)

        1.0 ... 1.543080635, -0.7615941560 ... 0.7615941560
```

For functions with discontinuities, evaluation over an interval may result in a union of intervals:

```
>> coth(-1 ... 1)

      RD_NINF ... -1.313035285 union 1.313035285 ... RD_INF
```

Example 2. Simplifications are implemented for arguments that are integer multiples of $i\,\pi/2$:

```
>> sinh(I*PI/2), cosh(40*I*PI), tanh(-10^100*I*PI),
   coth(-17/2*I*PI)

                          I, 1, 0, 0
```

Negative real numerical factors in the argument are rewritten via symmetry relations:

```
>> sinh(-5), cosh(-3/2*x), tanh(-x*PI/12), coth(-12/17*x*y*PI)

                 / 3 x \        / x PI \        / 12 x y PI \
   -sinh(5), cosh| --- |, - tanh| ---- |, - coth| --------- |
                 \  2  /        \  12  /        \     17    /
```

Example 3. The `expand` function implements the addition theorems:

```
>> expand(sinh(x + PI*I)), expand(cosh(x + y))

           -sinh(x), cosh(x) cosh(y) + sinh(x) sinh(y)
```

The `combine` function uses these theorems in the other direction, trying to rewrite products of hyperbolic functions:

```
>> combine(sinh(x)*sinh(y), sinhcosh)

                    cosh(x + y)   cosh(x - y)
                    ----------- - -----------
                         2             2
```

Example 4. Various relations exist between the hyperbolic functions:

```
>> csch(x), sech(x)

                            1        1
                         -------, -------
                         sinh(x)  cosh(x)
```

The function `expand` rewrites all functions in terms of `sinh` and `cosh`:

```
>> expand(tanh(x)), expand(coth(x))

                         sinh(x)  cosh(x)
                         -------, -------
                         cosh(x)  sinh(x)
```

Use `rewrite` to obtain a representation in terms of a specific target function:

```
>> rewrite(tanh(x)*exp(2*x), sinhcosh), rewrite(sinh(x), tanh)

                                                     / x \
                                               2 tanh| - |
         sinh(x) (cosh(2 x) + sinh(2 x))             \ 2 /
         -------------------------------, --------------
                     cosh(x)                         / x \2
                                             1 - tanh| - |
                                                     \ 2 /

>> rewrite(sinh(x)*coth(y), exp), rewrite(exp(x), coth)

              2      / exp(x)   exp(-x) \      / x \
        (exp(y)  + 1) | ------ - ------- |  coth| - | + 1
                     \    2        2    /      \ 2 /
        ---------------------------------, -------------
                         2                     / x \
                   exp(y)  - 1             coth| - | - 1
                                               \ 2 /
```

Example 5. The inverse functions are implemented by `arcsinh`, `arccosh` etc.:

```
>> sinh(arcsinh(x)), sinh(arccosh(x)), cosh(arctanh(x))

                   2     1/2              1
             x, (x  - 1)   , ---------------------
                                   1/2        1/2
                            (x + 1)    (1 - x)
```

Note that `arcsinh(sinh(x))` does not necessarily yield `x`, because `arcsinh` produces values with imaginary parts in the interval $[-\pi/2, \pi/2]$:

```
>> arcsinh(sinh(3)), arcsinh(sinh(1.6 + 100*I))

                         3, 1.6 - 0.5309649149 I
```

Example 6. Various system functions such as `diff`, `float`, `limit`, or `series` handle expressions involving the hyperbolic functions:

```
>> diff(sinh(x^2), x), float(sinh(3)*coth(5 + I))

                        2
            2 x cosh(x ), 10.01749636 - 0.0008270853591 I

>> limit(x*sinh(x)/tanh(x^2), x = 0)

                                    1

>> series((tanh(sinh(x)) - sinh(tanh(x)))/sinh(x^7), x = 0)

                                   2          4
                               29 x    1913 x       6
                - 1/30 + ----- - ------- + O(x )
                               756      75600

>> series(tanh(x), x = infinity)

        2         2         2         2          2
 1 - ------- + ------- - ------- + ------- - -------- +
            2         4         6         8          10
     exp(x)    exp(x)    exp(x)    exp(x)    exp(x)

      /     1    \
     O| -------- |
      |       12 |
      \ exp(x)   /
```

Changes:

- Floating point intervals are handled.
- `diff(tanh(x), x)` now returns `tanh(x)^2 + 1` instead of `1/cosh(x)^2`.

`slot` – method or entry of a domain or a function environment

`slot(d, "n")` returns the value of the slot named `"n"` of the object `d`.

`slot(d, "n", v)` creates or changes the slot `"n"`. The value `v` is assigned to the slot.

Call(s):

- `d :: n`
- `slot(d, "n")`
- `d :: n := v`
- `slot(d, "n", v)`
- `object :: dom`
- `slot(object, "dom")`

Parameters:

`d`	—	a domain or a function environment
`n`	—	the name of the slot: an identifier
`v`	—	the new value of the slot: an arbitrary MuPAD object
`object`	—	an arbitrary MuPAD object

Return Value: `slot(d, "n")` returns the value of the slot; `slot(d, "n", v)` returns the object `d` with the added or changed slot; `slot(object, "dom")` returns the domain type of the object.

Overloadable by: `d`

Related Functions: `DOM_DOMAIN`, `DOM_FUNC_ENV`, `domain`, `frame`, `funcenv`, `newDomain`

Details:

- The function `slot` is used for defining methods and entries of data types (domains) or for defining attributes of function environments. Such methods, entries, or attributes are called *slots*. They allow to overload system functions by user defined domains and function environments. See the "Background" section below for further information.

- The function `slot` is also used for accessing the value of an identifier in a `frame`. The values of identifiers in frames can be interpreted as slots of frames with the corresponding names. Complete information concerning the access of values of identifiers in frames can be found on the `frame` help page.

- Any MuPAD object has a special slot named `"dom"`. It holds the domain the object belongs to: `slot(object, "dom")` is equivalent to `domtype(object)`. The value of this special slot cannot be changed. Cf. example 1.

- Apart from the special slot `"dom"`, only domains and function environments may have further slots.

  The call `slot(d, "n")` is equivalent to `d::n`. It returns the value of the slot.

The call `slot(d, "n", v)` returns the object `d` with an added or changed slot `"n"` bearing the value `v`.

- For a *function environment* `d`, the call `slot(d, "n", v)` returns a *copy* of `d` with the changed slot `"n"`. The function environment `d` itself is not changed! Use the assignment `d := slot(d, "n", v)` to modify `d`. Cf. example 2.

  For a *domain* `d`, however, the call `slot(d, "n", v)` modifies `d` as a side-effect! This is the so-called “reference effect” of domains. Cf. example 3.

- If a non-existing slot is accessed, `FAIL` is returned as the value of the slot. Cf. example 4.

- The `::`-operator is a shorthand notation to access a slot.

  The expression `d::n`, when not appearing on the left hand side of an assignment, is equivalent to `slot(d, "n")`.

  The command `d::n := v` assigns the value `v` to the slot `"n"` of `d`. This assignment is almost equivalent to changing or creating a slot via `d:= slot(d, "n", v)`. Note the following subtle semantical difference between these assignments: in `d::n := v`, the identifier `d` is evaluated with level 1, i.e., the slot `"n"` is attached to the *value* of `d`. In `slot(d, "n", v)`, the identifier `d` is *fully evaluated*. See example 6.

- With `delete d::n` or `delete slot(d,"n")`, the slot `"n"` of the function environment or the domain `d` is deleted. Cf. example 5. The special slot `"dom"` cannot be deleted.

- The first argument of `slot` is not flattened. This allows to access the slots of expression sequences and `null()` objects. Cf. example 7.

- For domains, there is a special mechanism to create new values for slots on demand. If a non existing slot is read, the method `"make_slot"` of the domain is called in order to create the slot. If such a method does not exist, `FAIL` is returned. Cf. example 8.

- `slot` is a function of the system kernel.

Example 1. Every object has the slot `"dom"`:

```
>> slot(x, "dom") = domtype(x),
   slot(45, "dom") = domtype(45),
   slot(sin, "dom") = domtype(sin)

 DOM_IDENT = DOM_IDENT, DOM_INT = DOM_INT,

    DOM_FUNC_ENV = DOM_FUNC_ENV
```

Example 2. Here we access the existing `"float"` slot of the function environment `sin` implementing the sine function. The float slot is again a function environment and may be called like any MuPAD function. Note, however, the different functionality: in contrast to `sin`, the float slot always tries to compute a floating point approximation:

```
>> s := slot(sin, "float"): s(1) , sin(1)

                      0.8414709848, sin(1)
```

With the following command, `s` becomes the function environment `sin` apart from a changed `"float"` slot. The `slot` call has no effect on the original `sin` function because `slot` returns a copy of the function environment:

```
>> s := slot(sin, "float", x -> float(x - x^3/3!)):
   s(PI/3) = sin(PI/3), s::float(1) <> sin::float(1)

             1/2    1/2
            3      3
            ---- = ----, 0.8333333333 <> 0.8414709848
              2      2

>> delete s:
```

Example 3. If you are using the `slot` function to change slot entries in a domain, you must be aware that you are modifying the domain. This is in contrast to changing slots of function environments (see example 2):

```
>> old_one := slot(Dom::Float, "one")

                               1.0

>> newDomFloat := slot(Dom::Float, "one", 1):
   slot(newDomFloat, "one"), slot(Dom::Float, "one")

                              1, 1
```

We restore the original state:

```
>> slot(Dom::Float, "one", old_one): slot(Dom::Float, "one")

                               1.0

>> delete old_one, newDomFloat:
```

Example 4. The function environment `sin` does not contain a `"sign"` slot. So accessing this slot yields `FAIL`:

```
>> slot(sin, "sign"), sin::sign

                              FAIL, FAIL
```

Example 5. We define a function environment for a function computing the logarithm to the base `10`:

```
>> log10 := funcenv(x -> log(10, x)):
```

If the function `info` is to give some information about `log10`, we have to define the `"info"` slot for this function. For function environments, `slot` returns a copy of the original object, so the result of the `slot` call has to be assigned to `log10`:

```
>> log10 := slot(log10, "info",
                "log10 -- the logarithm to the base 10"):

>> info(log10)

 log10 -- the logarithm to the base 10
```

The `delete` statement is used for deleting a slot:

```
>> delete log10::info: info(log10)

 log10 -- a library procedure [try ?log10 for help]
```

It is not possible to delete the special slot `"dom"`:

```
>> delete log10::dom

 Error: Illegal argument [delete]

>> delete log10:
```

Example 6. Here we demonstrate the subtle difference between the `slot` function and the use of the `::`-operator in assignments. The following call adds a `"xyz"` slot to the domain `DOM_INT` of integer numbers:

```
>> delete b: d := b: b := DOM_INT: slot(d, "xyz", 42):
```

The slot `"xyz"` of `DOM_INT` is changed, because `d` is fully evaluated with the result `DOM_INT`. Hence, the slot `DOM_INT::xyz` is set to 42:

```
>> slot(d, "xyz") , slot(DOM_INT, "xyz")
```

42, 42

Here is the result when using the `::`-operator: `d` is only evaluated with level 1, i.e., it is evaluated to the identifier `b`. However, there is no slot `b::xyz`, and an error occurs:

```
>> delete b: d := b: b := DOM_INT: d::xyz := 42
```

```
 Error: Unknown slot "d::xyz" [slot]
```

```
>> delete b, d:
```

Example 7. The first argument of `slot` is not flattened. This allows access to the slots of expression sequences and `null()` objects:

```
>> slot((a, b), "dom"), slot(null(), "dom")
```

DOM_EXPR, DOM_NULL

Example 8. We give an example for the use of the function `make_slot`. The element `undefined` of the domain `stdlib::Undefined` represents an undefined value. Any function `f` should yield `f(undefined) = undefined`. Inside the implementation of `stdlib::Undefined`, we find:

```
>> undef := newDomain("stdlib::Undefined"):
   undefined := new(undef):
   undef::func_call := proc() begin undefined end_proc;
   undef::make_slot := undef::func_call:
```

The following mechanism takes place automatically for a function `f` that is overloadable by its first argument: in the call `f(undefined)`, it is checked whether the slot `undef::f` exists. If this is not the case, the `make_slot` function creates this slot "on the fly", producing the value `undefined`. Thus, via overloading, `f(undefined)` returns the value `undefined`.

Example 9. The following example is rather advanced and technical. It demonstrates overloading of the `slot` function to implement slot access and slot assignments for other objects than domains (`DOM_DOMAIN`) or function environments (`DOM_FUNC_ENV`). The following example defines the slots `"numer"` and `"denom"` for rational numbers. The domain `DOM_RAT` of such numbers does not have slots `"numer"` and `"denom"`:

```
>> domtype(3/4)
```

```
>> slot(3/4, "numer");

 Error: Unknown slot "(3/4)::numer" [slot]
```

We can change DOM_RAT, however. For this, we have to unprotect DOM_RAT temporarily:

```
>> unprotect(DOM_RAT):
   _assign(DOM_RAT::slot,
     proc(r : DOM_RAT, n : DOM_STRING, v=null(): DOM_INT)
         local i : DOM_INT;
     begin
         i := contains(["numer", "denom"], n);
         if i = 0 then
            error("Unknown slot \"".expr2text(r)."::".n."\"")
         end;
         if args(0) = 3 then
            subsop(r, i = v)
         else
            op(r, i)
         end
     end_proc):
```

Now, we can access the operands of rational numbers, which are the numerator and the denominator respectively, via our new slots:

```
>> slot(3/4, "numer"), (3/4)::numer,
   slot(3/4, "denom"), (3/4)::denom

                          3, 3, 4, 4

>> a := 3/4: slot(a, "numer", 7)

                             7/4

>> a::numer := 11: a

                             11/4
```

We restore the original behaviour:

```
>> delete DOM_RAT::slot, a: protect(DOM_RAT, Error):
```

Background:

- Overloading of system functions by domain elements is typically implemented as follows. If a library function `f`, say, is to be overloadable by user defined data types, a code segment as indicated by the following lines is appropriate. It tests whether the domain `x::dom` of the argument `x` contains a method `f`. If this is the case, this domain method is called:

```
f:= proc(x)
    begin
      // check if f is overloaded by x
      if x::dom::f <> FAIL then
           // use the method of the domain of x
           return(x::dom::f(args()))
      else
           // execute the code for the function f
      endif
    end_proc:
```

- By overloading the function `slot`, slot access and slot assignment can be implemented for other objects than domains or function environments. Cf. example 9.
- In principle, the name `n` of a slot may be an arbitrary MuPAD object. Note, however, that the `::`-operator cannot access slots defined by `slot(d, n, v)` if the the name `n` is not a string.
- Strings may be used in conjunction with the `::`-operator: the calls `d::"n"` and `d::n` are equivalent.

Changes:

- `slot` is now also used to access the value of an identifier in a `frame`.

`solve` – solve equations and inequalities

`solve(eq, x)` returns the set of all complex solutions of an equation or inequality `eq` with respect to `x`.

`solve(system, vars)` solves a system of equations for the variables `vars`.

`solve(eq, vars)` is equivalent to `solve([eq], vars)`.

`solve(system, x)` is equivalent to `solve(system, [x])`.

`solve(eq)` without second argument is equivalent to `solve(eq, S)` where `S` is the set of all indeterminates in `eq`. The same holds for `solve(system)`.

Call(s):

- `solve(eq, x <, options>)`
- `solve(eq, vars <, options>)`
- `solve(eq <, options>)`
- `solve(system, x <, options>)`
- `solve(system, vars <, options>)`
- `solve(system <, options>)`
- `solve(ODE)`
- `solve(REC)`

Parameters:

`eq`	—	a single equation or an inequality of type `"_equal"`, `"_less"`, `"_leequal"`, or `"_unequal"`. Also an arithmetical expression is accepted and regarded as an equation with vanishing right hand side.
`x`	—	the indeterminate to solve for: an identifier or an indexed identifier
`vars`	—	a non-empty set or list of indeterminates to solve for
`system`	—	a set, list, array, or table of equations and/or arithmetical expressions. Expressions are regarded as equations with vanishing right hand side.
`ODE`	—	an ordinary differential equation: an object of type `ode`.
`REC`	—	a recurrence equation: an object of type `rec`.

Options:

MaxDegree = n	— do not use explicit formulas involving radicals to solve polynomial equations of degree larger than `n`. The default value of the positive integer `n` is 2.
BackSubstitution = b	— do or do not perform back substitution when solving algebraic systems; `b` must be `TRUE` or `FALSE`. The default value is `TRUE`.
Multiple	— returns the solution set as an object of type `Dom::Multiset`, indicating the multiplicity of polynomial roots. This option is only allowed for polynomial equations and polynomial expressions.
PrincipalValue	— return only one solution as a set with one element
Domain = d	— return the set of all solutions that are elements of `d`. `d` must represent a subset of the complex numbers (for example, the reals or the integers) or must be a domain over which polynomials can be factored, e.g., a finite field. In the latter case, this option is only allowed for polynomial equations. If this option is missing, all solutions in the set of complex numbers are returned.
IgnoreProperties	— return also solutions that are not consistent with the properties of the variable `x`.
IgnoreSpecialCases	— If a case analysis becomes necessary, ignore all cases which suppose some parameter in the equation to be an element of a fixed finite set.
DontRewriteBySystem	— Do not try to solve an equation by transforming it into an equivalent system of equations. This option decreases the running time, at the cost of not being able to solve certain equations.

Return Value: `solve(eq, x)` returns an object that represents a mathematical set (see "Details") . A call to `solve` returns a set of lists if one of the arguments is a set or a list, or if the first argument is an array or a table, or if the second argument is missing. Each list consists of equations, where the left hand side contains a variable to be solved for. `solve` may also return an expression of the form `x in S`, where `x` is one of the variables to solve for, and `S` is some set.

Overloadable by: `eq`

Side Effects: `solve` reacts to properties of identifiers.

Related Functions: `linsolve`, `numeric::linsolve`, `numeric::solve`, `RootOf`, `solvers`

Details:

- The `solve` routine provides a unified interface to a variety of specialized solvers. See `?solvers` for an overview.

- If no indeterminates are specified, the set of all indeterminates appearing in `eq` (or `system`, respectively) is used. Indeterminates are identifiers (except mathematical constants such as `PI`, `EULER` etc.) and indexed identifiers. Indeterminates that appear only inside function names or indices are discarded. Cf. example 8.

- If a list of indeterminates to solve for is specified, the components of the resulting solution vectors are sorted according to the specified ordering of the indeterminates. If indeterminates are specified by a set, some ordering of the indeterminates is chosen internally.

- The sets returned by `solve` can be of many different types (an overview is given in the "Background" section below). You can never foresee what type of set will be returned. However, they have a unified interface of functions that may be applied to all of them. These functions include the set-theoretic operations `intersect`, `union`, and `minus`. Cf. example 2. Further, pointwise defined arithmetical operations `+`, `*` etc. can be applied. The function `solvelib::getElement` serves for extracting elements. The function `solvelib::isFinite` tests whether the solution set returned by `solve` is finite.

- `solve(eq, x)` returns only those solutions that are consistent with the properties of `x`. Cf. example 9. When solving a system of equations for several variables, the properties of the variables to solve for may, but need not be taken into account. If the option *`IgnoreProperties`* is given, properties are not used.

- For inequalities, it is assumed that the variable(s) to solve for (but not necessarily the free parameters) are real.

- An inequality `a<=b` or `a<b` can only hold if both sides are real. In particular, `a=b` does not imply `a<=b`.

- Since `solve` may be overloaded, special domains for equations of special kinds can be written. The MuPAD library itself uses this feature in the case of differential equations (see `ode`) and recurrence equations (see `rec`). Thus, `solve` provides an interface for solving differential and recurrence equations. See the helppages of `ode` and `rec` for examples.

- The command `float(hold(solve)(equations, indeterminates <, options>))` yields a numerical solution. It is equivalent to the call `numeric::solve(equations, indeterminates <, options>)`. See the help page of `numeric::solve` for the available options. In particular, starting points and search ranges for the numerical search can be specified. Note that for non-polynomial equations, only a single numerical solution is searched for. Cf. example 12.

  In contrast to `solve`, `numeric::solve` does not react to properties of identifiers set via `assume`. NOTE

Option <`Multiple`>:

- With this option, `solve` returns a set of type `Dom::Multiset`.
- Trying to solve the zero polynomial with option *`Multiple`* causes an error since infinite multisets cannot be represented in MuPAD .
- If the solution is of type `RootOf`, this option is ignored.

Option <`PrincipalValue`>:

- With this option, only one solution is returned even if several solutions exist. If the equation does not have a solution, an empty set is returned.
- If no single element of the set of solutions can be found, the result is a symbolic call of `solve`. In particular, this is the case if the set of solutions is piecewise defined and no element is common to all cases.
- This option may also be used, when equations are to be solved for several variables. In this case, a set containing only one list (representing a solution vector) is returned.

Option <`MaxDegree = n`>:

- This option enables/disables the use of explicit formulas for the zeroes of polynomials; other methods such as factorization are always applied. For polynomial equations, the given maximal degree `n` refers to the factors of the polynmials and not to the input polynomial.
- For polynomials of degree larger than 4, no explicit formulas exist. It makes no difference whether *`MaxDegree`* is set to 4 or to a higher value.

Option <`BackSubstitution = b`>:

- An object of type `"RootOf"` is never substituted into a variable. Hence, even if *`BackSubstitution`* is set to `TRUE`, the solution for one variable may be the set of roots of a polynomial depending on another variable.

Option <`Domain = d`>:

- Equations and systems in more than one variable cannot be solved over domains.

- Two kinds of Domains `d` are possible: subsets of `C_` and domains where polynomials can be factored (for polynomial equations only).

- A subset of `C_` can be any kind of set returned by `solve` (see the "Background" section). Instead of `C_`, `R_`, `Q_`, and `Z_`, the corresponding domains of the domains package `Dom::Complex`, `Dom::Real`, `Dom::Rational`, and `Dom::Integer` may be used.

- In addition, domains `d` may overload this part of the solver as follows: equations and systems are solved by the `"domsolve"` method; if that one does not exist, equations are solved by the `"solve_eq"` method, while systems are not allowed as arguments in this case; finally, if that method also does not exist, polynomials are solved by the `"solve_poly"` method, while any first argument to `solve` that cannot be converted to a polynomial is illegal. The `"domsolve"` method is called `domsolve(eq, var, options)`, where `eq` and `var` have the same meaning as for `solve`, and `options` is a table of options. The methods `"solve_eq"` and `"solve_poly"` have the same calling syntax, but with `eq` being an arithmetical expression or a polynomial, respectively, in this cases.

Option <`IgnoreSpecialCases`>:

- This option makes `solve` apply some kind of heuristics in order to reduce the number of branches in piecewise defined objects: all equalities are assumed to be `FALSE`, except those that the property mechanism can prove to be `TRUE`; for example, all denominators that are not provably zero are assumed to be nonzero. This option tends to decrease the number of piecewise defined objects in results considerably.

Option <`DontRewriteBySystem`>**:**

- Rewriting by an equivalent system is mainly used for solving equations with nested roots. Since solving the resulting system tends to take much time, this option may be used to make the solver less powerful but quicker.

Example 1. Usually, a set of type `DOM_SET` is returned if an equation has a finite number of solutions:

```
>> solve(x^4 - 5*x^2 + 6*x = 2, x)

                          1/2          1/2
                    {1, 3    - 1, - 3    - 1}
```

Example 2. The solution set may also be an infinite discrete set:

```
>> S := solve(sin(x*PI/7) = 0, x)

                     { 7*X2 |  X2 in Z_ }
```

To pick out the solutions in a certain finite interval, just intersect the solution set with the interval:

```
>> S intersect Dom::Interval(-22, 22)

                 {-21, -14, -7, 0, 7, 14, 21}

>> delete S:
```

Example 3. Sometimes, a call to `simplify` or `normal` may be necessary to obtain a simple result.

```
>> solve(cos(PI*x) = 0)

          x in { 1/PI*(1/2*PI + X4*PI) |  X4 in Z_ }

>> normal(%)

                 x in { X4 + 1/2 |  X4 in Z_ }
```

Example 4. The solution set of an inequality is usually an interval or a union of intervals:

```
>> solve(x^2 > 5, x)

         ]5^(1/2), infinity[ union ]-infinity, -5^(1/2)[
```

Example 5. For all but two numbers x, their square does not equal 7:

```
>> solve(x^2 <> 7, x)

                              1/2      1/2
                     C_ minus {7    , - 7    }
```

Example 6. In some cases, the solution is a union of an infinite family of sets. Such infinite unions are represented using `solvelib::Union`:

```
>> solve(x^x = 3, x)

 solvelib::Union(solve(x ln(x) = 2 I X5 PI + ln(3), x), X5, Z_)
```

Example 7. Piecewise defined objects occur if an equation contains symbolic parameters in addition to the variable to solve for. The well-known solution formula for the quadratic equation $ax^2 + bx + c = 0$ is only valid for $a <> 0$; otherwise the equation reduces to $bx = -c$, where again two cases $b <> 0$ and $b = 0$ must be treated separately. Finally, if $b = 0$, the equation $c = 0$ may be true — then, every x is a solution — or false — then, no x is a solution.

```
>> S := solve(a*x^2 + b*x + c, x)

          /
          |
          |
          |
 piecewise| C_ if a = 0 and b = 0 and c = 0,
          \

    {} if a = 0 and b = 0 and c <> 0,

                             {                   2 1/2
                             {   b   (- 4 a c + b )
                             { - - - -----------------
    {   c }                  {   2           2
```

```
{ - - } if a = 0 and b <> 0, { ----------------------,
{   b }                      {            a

                 2 1/2 }            \
  b   (- 4 a c + b )    }           |
- - + ----------------- }           |
  2           2         }           |
----------------------- } if a <> 0 |
          a             }           /
```

You might want to make additional assumptions and re-evaluate the result:

```
>> assume(a <> 0): S
```

```
{         2          1/2    2          1/2     }
{   b   (b  - 4 a c)      (b  - 4 a c)      b }
{ - - - ---------------  --------------- - - }
{   2          2                  2          2 }
{ ---------------------, ------------------- }
{           a                     a          }
```

```
>> delete S: unassume(a):
```

Example 8. If no indeterminates are specified, the set of all indeterminates in the equation is used:

```
>> solve(x^2 = 3)
```

```
              1/2             1/2
        {[x = 3   ], [x = - 3   ]}
```

Indeterminates are only searched for outside operators and indices. Hence, neither `f` nor `y` is an indeterminate of the following equation:

```
>> solve(f(x[y]) = 7)
```

```
        solve({f(x[y]) = 7}, [x[y]])
```

Example 9. If the unknown to solve carries a mathematical property, only the solutions compatible with that property are returned. In the following, `x` is assumed to be a positive number (implying that `x` is real):

```
>> assume(x, Type::Positive): solve(x^4 = 1, x)
```

```
                     {1}
```

Without a property, all complex solutions are returned:

```
>> unassume(x): solve(x^4 = 1, x)

                        {-1, 1, -I, I}
```

Example 10. Using the option *Multiple*, the multiplicity of zeroes of polynomials becomes visible. Below, we see that $x = -1$ is a double zero of x^3+2x^2+x, while $x = 0$ has only multiplicity one:

```
>> solve(x^3 + 2*x^2 + x, x, Multiple)

                       {[0, 1], [-1, 2]}
```

Example 11. If *BackSubstitution* is set to `FALSE`, the solution for a variable `y` may contain another variable `x` to solve for, but only if `x` appears on the right of `y` in the list of indeterminates.

```
>> solve({x^2 + y = 1, x - y = 2}, [y, x], BackSubstitution = FALSE)

 { --                     1/2       --
 { |                    13           |
 { |  y = x - 2, x = - ----- - 1/2  |,
 { --                     2         --

    --                   1/2       -- }
    |                  13           | }
    |  y = x - 2, x = ----- - 1/2  | }
    --                   2         -- }

>> solve({x^2 + y = 1, x - y = 2}, {x, y})

 { --         1/2                 1/2        --
 { |        13                  13            |
 { |  x = - ----- - 1/2, y = - ----- - 5/2  |,
 { --         2                   2          --

    --        1/2               1/2        -- }
    |       13                13            | }
    |  x = ----- - 1/2, y = ----- - 5/2   | }
    --        2                 2          -- }
```

Although *BackSubstitution* is switched on in the following example, the solution for `y` still depends on `x` because variables are never substituted by objects of type `"RootOf"`.

```
>> solve({x^2 + y = 1, x - y = 2}, [y, x], MaxDegree = 1)

                                              2
          {[y = x - 2, x = RootOf(X7 + X7  - 3, X7)]}
```

Example 12. MuPAD's solver does not find an exact symbolic solution of the following equation:

```
>> solve(2^x = x^2, x)

                          x    2
                   solve(2  - x  = 0, x)
```

Applying `float` invokes the numerical solver:

```
>> float(%)

                       {-0.766664696}
```

Avoiding the overhead of the symbolic solver, the numerical solver is called directly via the following command:

```
>> float(hold(solve)(2^x - x^2, x))

                       {-0.766664696}
```

When applied to a non-polynomial equation, the numerical solver returns at most *one* solution, even if there are more. Search ranges can be specified to find other solutions:

```
>> float(hold(solve)(2^x - x^2, x = 0..3))

                           {2.0}
```

As an alternative to `float(hold(solve)(...))`, the numerical solver `numeric::solve` can be called directly:

```
>> numeric::solve(2^x - x^2, x = 3..6)

                           {4.0}
```

For polynomial equations, the numerical solver returns *all* complex solutions:

```
>> solve(x^4 + x^3 = 3*x, x)

                                 2     3
              {0} union RootOf(X8  + X8  - 3, X8)

>> float(%)
```

```
{0.0} union {- 1.087279705 + 1.171312111 I,

   - 1.087279705 - 1.171312111 I, 1.17455941}

>> eval(%)

 {1.17455941, 0.0, - 1.087279705 + 1.171312111 I,

   - 1.087279705 - 1.171312111 I}
```

In general, we recommend not to use intermediate symbolic results if numerical approximations are desired. Note that symbolic preprocessing may be time consuming, and the numerical evaluation of symbolic results may be numerically unstable. A direct call to the numerical solver `numeric::solve` avoids such problems.

Background:

- The following types of sets may be returned by `solve`:

  - finite sets (type `DOM_SET`);
  - symbolic calls to the function `solve`;
  - zero sets of polynomials (type `RootOf`). `solve` returns this type if it is not able or, by virtue of the option `MaxDegree`, not allowed to solve the equation explicitly in terms of radicals;
  - set-theoretic expressions (types `"_union"`, `"_intersect"`, and `"_minus"`);
  - symbolic calls of `solvelib::Union`. These represent unions over parametrized systems of sets;
  - the sets $\mathbb{C}, \mathbb{R}, \mathbb{Q}$, and $\mathbb{Z}$ (type `solvelib::BasicSet`);
  - intervals (type `Dom::Interval`);
  - image sets of functions (type `Dom::ImageSet`);
  - piecewise defined objects, where every branch defines some set of one of the mentioned types (type `piecewise`).

Changes:

- New options `DontRewriteBySystem` and `IgnoreProperties` have been introduced.

- `solve` now reacts to `"solve_eq"` and `"solve_poly"` slots of domains.

`solvers` – an overview of **MuPAD**'s solvers

Besides the general `solve` command, MuPAD provides a variety of specialized solvers for special types of equations. The specialized solvers only handle a subclass of problems, but are more efficient than the general `solve`.

Call(s):

- `detools::pdesolve(..)`
- `linalg::matlinsolve(..)`
- `linalg::matlinsolveLU(..)`
- `linalg::vandermondeSolve(..)`
- `linsolve(..)`
- `numeric::linsolve(..)`
- `numeric::matlinsolve(..)`
- `numeric::fsolve(..)`
- `numeric::odesolve(..)`
- `numeric::odesolve2(..)`
- `numeric::odesolveGeometric(..)`
- `numeric::polyroots(..)`
- `numeric::polysysroots(..)`
- `numeric::realroot(..)`
- `numeric::realroots(..)`
- `numeric::solve(..)`
- `numlib::mroots(..)`
- `numlib::lincongruence(..)`
- `numlib::msqrts(..)`
- `polylib::realroots(..)`
- `series(..)`
- `solve(..)`

Details:

- The following types of equations can be solved:

type of equation	available solvers

system of linear equations	`solve`
	`linsolve`
	`linalg::matlinsolve`
	`linalg::matlinsolveLU`
	`linalg::vandermondeSolve`
	`numeric::linsolve`
	`numeric::matlinsolve`
	`numeric::solve`

univariate polynomial equation	`solve`
	`polylib::realroots`
	`numeric::solve`
	`numeric::polyroots`

bivariate polynomial equation	`solve` `series`

system of polynomial equations	`solve`
	`numeric::solve`
	`numeric::polysysroots`

arbitrary univariate equation	`solve`
	`numeric::solve`
	`numeric::realroot`
	`numeric::realroots`

system of arbitrary equations	`solve`
	`numeric::solve`
	`numeric::fsolve`

inequalities	`solve`

system of ordinary differential equations	`solve`
	`numeric::odesolve`
	`numeric::odesolve2`
	`numeric::odesolveGeometric`

recurrence equation	`solve`
partial differential equations	`detools::pdesolve`

congruence	`numlib::lincongruence`
	`numlib::mroots`
	`numlib::msqrts`

- `detools::pdesolve` allows to solve partial differential equations. This first version of the solver is not yet very powerful; essentially only the method of characteristics has been implemented for quasi-linear first order equations.

- `linalg::vandermondeSolve` is the recommended solver for systems of linear equations with a coefficient matrix of Vandermonde type.

- `linalg::matlinsolve` solves systems of linear equations given by a coefficient matrix. The coefficient domain may be an arbitrary field.

  For coefficients of basic type such as expressions, integers, rationals, floating point numbers etc. we recommend to use `numeric::matlinsolve` instead.

- `linalg::matlinsolveLU` solves systems of linear equations over arbitrary fields with a coefficient matrix given by an LU-decomposition.

- `linsolve` is the recommended solver for systems of linear equations over arbitrary non-elementary coefficient domains. If the coefficients are of basic type such as expressions, integers, rationals, floating point numbers etc., then we recommend to use `numeric::linsolve` instead.

- `numeric::linsolve` is a fast numerical solver for systems of linear equations. It is also capable of computing exact symbolic solutions, if the coefficients are MuPAD expressions. If the solution is to be computed over some non-basic coefficient domain, then `linsolve` must be used.

- `numeric::matlinsolve` is a fast numerical solver for systems of linear equations given by a coefficient matrix. It is also capable of computing exact symbolic solutions, if the coefficients are MuPAD expressions. If the solution is to be computed over some non-basic coefficient domain, then `linalg::matlinsolve` must be used.

- `numeric::fsolve` is the general numerical solver for systems of arbitrary equations. It returns only one solution.

- `numeric::odesolve` is the numerical solver for initial value problems of systems of ordinary differential equations.

- `numeric::odesolve2` encapsulates `numeric::odesolve` in a function. This provides a convenient interface to `numeric::odesolve`.

- `numeric::odesolveGeometric` is a specialized solver for ODEs on homogeneous manifolds embedded in a matrix space.

- `numeric::polyroots` computes numerical approximations of all roots of a single univariate polynomial.

- `numeric::polysysroots` computes all roots of a system of multivariate polynomial equations. This is a hybrid algorithm using symbolic Gröbner techniques and numerical post-processing.

- `numeric::realroot` computes a single real root of an arbitrary real equation.

- `numeric::realroots` isolates all real roots of a single arbitrary real equation via interval arithmethic.

- `numeric::solve` provides a common interface to the solvers `numeric::fsolve`, `numeric::linsolve`, `numeric::polyroots`, and `numeric::polysysroots`. It determines the type of the input equation(s) and calls one of these routines.

- `numlib::lincongruence` solves linear congruences.
- `numlib::mroots` solves polynomial congruences.
- `numlib::msqrts` computes the modular square roots of a number.
- `polylib::realroots` provides exact isolation of all real roots of a single univariate real polynomial.
- `series(RootOf(p, y), x = x0)` computes all series solutions around $x = x0$ of the polynomial equation $p(y) = 0$.
- `solve` is the general solver for equations and systems of equations as well as inequalities; it also provides an interface to solving ordinary differential equations (see `ode`) and recursion relations (see `rec`).

  The library `solvelib` provides various utilities for handling the output of `solve`.

`sort` – sort a list

`sort(list)` returns a sorted copy of the list.

Call(s):

- `sort(list <, f>)`

Parameters:

`list`	—	a list of arbitrary MuPAD objects
`f`	—	a procedure defining the ordering

Return Value: a list.

Overloadable by: `list`

Related Functions: `sysorder`

Details:

- `sort` sorts the list in "ascending order".
- If no procedure `f` is specified by the user, lists are sorted as follows:
  - A list with real numbers of syntactical type `Type::Real` is sorted numerically.
  - A list of character strings is sorted lexicographically.

- In all other cases, the list is sorted according to the system's internal order, i.e., `sort(list)` is equivalent to `sort(list, sysorder)`.

  The internal order is a well-ordering. It is not session dependent, but may differ between different MuPAD versions. No special features of the internal sorting mechanism should be assumed by the user. Such sorting may be useful to produce a unique representation of lists.

- When strings are compared, capital letters are sorted in front of small letters. E.g., `"Z"` is smaller than `"abc"`.

- Sets and tables do not have a unique internal order (`sysorder`). Consequently, sorting does not lead to a unique ordering, if elements of the list are sets or tables, or contain sets or tables as (sub)operands. Cf. example 2.

- A procedure `f` may be specified to define the sorting criteria. It is used to compare the ordering of pairs of list elements and is called in the form `f(x, y)` with elements `x`, `y` from the list. It must return a Boolean expression that can be evaluated to either `TRUE` or `FALSE`. `TRUE` indicates that `x` is to be sorted left of `y`. Consequently, the elements of the ordered list `L := sort(list, f)` satisfy `bool(f(L[i], L[j])) = TRUE` for `i < j`.

  If the ordering provided by `f` is not a well-ordering, sorting is not 'stable' and elements with the same order may be swapped.

- The mean run time for sorting n elements is $O(n \log n)$.

- `sort` is a function of the system kernel.

Example 1. Real numbers of syntactical type `Type::Real` are sorted numerically:

```
>> sort([4, -1, 2/3, 0.5])

                    [-1, 0.5, 2/3, 4]
```

Strings are sorted lexicographically:

```
>> sort(["chip", "alpha", "Zip"])

                 ["Zip", "alpha", "chip"]
```

Other types of objects are sorted according to their internal ordering. This also holds for lists with elements of different types:

```
>> sort([4, -1, 2/3, 0.5, "alpha"])

                ["alpha", -1, 4, 2/3, 0.5]

>> sort([4, -1, 2/3, 0.5, I])

                   [-1, 4, 2/3, I, 0.5]
```

Example 2. There is no unique internal order for sets and tables:

```
>> sort([{1}, {2}]) <> sort([{2}, {1}])

                  [{1}, {2}] <> [{2}, {1}]

>> sort([table("a" = 42), table("a" = 43)]) <>
   sort([table("a" = 43), table("a" = 42)])

 -- table(       table(      --
 |    "a" = 42 ,   "a" = 43   | <>
 -- )            )           --

    -- table(       table(      --
    |    "a" = 43 ,   "a" = 42   |
    -- )            )           --
```

Example 3. The following list is sorted according to a user-defined criteron:

```
>> sort([-2, 1, -3, 4], (x, y) -> abs(x) < abs(y))

                       [1, -2, -3, 4]
```

Background:

- A variant of the Quicksort algorithm is used.

`sparsematrix` – create a sparse matrix or a sparse vector

`sparsematrix(m, n, [[a11, a12, ...], [a21, a22, ...], ...])` returns an $m \times n$ matrix of the domain type `Dom::SparseMatrix()`.

`sparsematrix(m, 1, [a1, a2, ...])` returns an $m \times 1$ column vector of the domain type `Dom::SparseMatrix()`.

`sparsematrix(1, n, [a1, a2, ...])` returns an $1 \times n$ row vector of the domain type `Dom::SparseMatrix()`.

Call(s):

- `sparsematrix(ListOfRows)`
- `sparsematrix(List)`
- `sparsematrix(Array)`

- sparsematrix(Matrix)
- sparsematrix(m, n)
- sparsematrix(m, n, ListOfRows)
- sparsematrix(m, n, Table)
- sparsematrix(m, n, [(i1, j1) = value1, (i2, j2) = value2, ...])
- sparsematrix(m, n, List, *Diagonal*)
- sparsematrix(m, n, List, *Banded*)
- sparsematrix(m, n, f)
- sparsematrix(m, n, g, *Diagonal*)
- sparsematrix(1, n, List)
- sparsematrix(1, n, Table)
- sparsematrix(1, n, [j1 = value1, j2 = value2, ...])
- sparsematrix(m, 1, List)
- sparsematrix(m, 1, Table)
- sparsematrix(m, 1, [i1 = value1, i2 = value2, ...])

Parameters:

ListOfRows	—	a nested list of rows, each row being a list of arithmetical expressions
List	—	a list of arithmetical expressions
Array	—	a one- or two-dimensional array
Matrix	—	a matrix, i.e., an object of a data type of category Cat::Matrix
Table	—	a table of matrix components
m	—	the number of rows: a positive integer
n	—	the number of columns: a positive integer
f	—	a function or a functional expression of two arguments
g	—	a function or a functional expression of one argument
i1, i2 ...	—	row indices: integers between 1 and m
j1, j2 ...	—	column indices: integers between 1 and m
value1, value2 ...	—	matrix entries: arithmetical expressions

Options:

Diagonal	—	create a diagonal matrix
Banded	—	create a banded Toeplitz matrix

Return Value: a matrix of the domain type Dom::SparseMatrix().

Related Functions: array, DOM_ARRAY, Dom::Matrix, Dom::SparseMatrix, matrix

Details:

- `sparsematrix` is equivalent to `Dom::SparseMatrix()`.

- `sparsematrix` creates matrices and vectors with a sparse internal representation. A column vector is represented as an $m \times 1$ matrix.. A row vector is represented as a $1 \times n$ matrix.

  Matrix and vector components must be arithmetical expressions (numbers and/or symbolic expressions). If sparse matrices over special component rings are desired, use the domain constructor `Dom::SparseMatrix` with a suitable component ring.

- Arithmetical operations with sparse matrices can be performed by using the standard arithmetical operators of MuPAD.

  E.g., if `A` and `B` are two matrices defined by `sparsematrix`, then `A + B` computes the sum and `A * B` computes the product of the two sparse matrices, provided that the dimensions are appropriate.

  Similarly, `A^(-1)` or `1/A` computes the inverse of a square sparse matrix `A` if it can be inverted. Otherwise, `FAIL` is returned.

  Cf. example 1.

- Many system functions accept sparse matrices as input, such as `map`, `subs`, `has`, `zip`, `conjugate`, `norm` or `exp`. Cf. example 4.

- Most of the functions in MuPAD's linear algebra package `linalg` work with sparse matrices. For example, to compute the determinant of a square sparse 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.

  For numerical matrix computations, the corresponding functions of the `numeric` package accept sparse matrices.

- Sparse matrix components can be extracted by the usual index operator `[ ]`, which also works for lists, arrays, and tables. The call `A[i, j]` extracts the sparse matrix component in the i-th row and the j-th column.

  Assignments to matrix components are performed similarly. The call `A[i, j] := c` replaces the matrix component in the i-th row and the j-th column of A by `c`.

  If one of the indices is not in its valid range, an error message is issued.

  The index operator also extracts submatrices. The call `A[r1..r2, c1..c2]` creates the submatrix of A comprising the rows with the indices $r_1, r_1 + 1, \ldots, r_2$ and the columns with the indices $c_1, c_1 + 1, \ldots, c_2$ of A.

  Cf. examples 3 and 5.

- `sparsematrix(ListOfRows)` creates an $m \times n$ sparse 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 non-zero.

  If a row has less than n entries, the remaining entries in the corresponding row of the sparse matrix are regarded as zero. Cf. example 7.

- `sparsematrix(List)` creates an $m \times 1$ column vector with components taken from the non-empty list, where m is the number of entries of `List`. Cf. example 5.

- `sparsematrix(Array)` or `sparsematrix(Matrix)` create a new sparse matrix with the same dimension and the components of `Array` or `Matrix`, respectively. The array must not contain any uninitialized entries. If `Array` is one-dimensional, the result is a column vector. Cf. example 8.

- The call `sparsematrix(m, n)` returns the $m \times n$ zero sparse matrix.

- `sparsematrix(m, n, ListOfRows)` creates an $m \times n$ sparse 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 sparse matrix.

  If a row has less than `n` entries, the remaining components of the corresponding row of the sparse matrix are regarded as zero. If there are less than `m` rows, the remaining lower rows of the sparse matrix are filled with zeroes. Cf. example 7.

- `sparsematrix(m,n,Table)` creates an $m \times n$ sparse matrix with components taken from the table `Table`. The table entries `Table[i,j]` with positive integer values of `i` and `j` define the corresponding entries of the sparse matrix. Zero entries need not be specified in the table. This way, sparse table input can be used to create the matrix.

  For large sparse matrices, the fastest way of creation is the generation of an empty table that is filled by indexed assignments and then passed to `sparsematrix`. Alternatively, one may first create an empty sparse matrix via `sparsematrix(m, n)` and then fill in the non-zero entries via indexed assigments. Note that the indexed assignment to a sparse matrix is somewhat slower than the indexed assignment to a table.

- `sparsematrix(m, n, [(i1, j1) = value1, (i2, j2) = value2, ...])` is a further way to create a sparse matrix specifying only the non-zero entries `A[i1, j1] = value1`, `A[i2, j2] = value2` etc. The ordering of the entries in the input list is irrelevant.

- `sparsematrix(m, n, f)` returns the sparse matrix whose (i, j)-th component is the return value of the function call `f(i,j)`. The row index i runs from 1 to m and the column index j from 1 to n. Cf. example 9.

- `sparsematrix(m, 1, List)` returns the $m \times 1$ column vector with components taken from `List`. The list `List` must have no more than `m` entries. If there are fewer entries, the remaining vector components are regarded as zero. Cf. example 5.

- `sparsematrix(m, 1, Table)` returns the $m \times 1$ column vector with components taken from `Table`. The table `Table` must have no more than `m` entries. If there are fewer entries, the remaining vector components are regarded as zero. Cf. example 6.

- `sparsematrix(m, 1, [i1 = value1, i2 = value2, ...])` provides a way to create a sparse column vector specifying only the non-zero entries `A[i1] = value1`, `A[i2] = value2` etc. The ordering of the entries in the input list is irrelevant.

- `sparsematrix(1, n, List)` returns the $1 \times n$ row vector with components taken from `List`. The list `List` must not have more than `n` entries. If there are fewer entries, the remaining vector components are regarded as zero. Cf. example 5.

- `sparsematrix(1, n, Table)` returns the $1 \times n$ row vector with components taken from `Table`. The table `Table` must not have more than `n` entries. If there are fewer entries, the remaining vector components are regarded as zero. Cf. example 6.

- `sparsematrix(1, n, [j1 = value1, j2 = value2, ...])` provides a way to create a sparse row vector specifying only the non-zero entries `A[j1] = value1`, `A[j2] = value2` etc. The ordering of the entries in the input list is irrelevant.

Option <`Diagonal`>:

- With this option, diagonal matrices can be created with diagonal elements taken from a list, or computed by a function or a functional expression.

- `sparsematrix(m, n, List, `*`Diagonal`*`)` creates the $m \times n$ diagonal matrix whose diagonal elements are the entries of `List`. Cf. example 10.

  `List` must have no more than $\min(m, n)$ entries. If it has fewer elements, the remaining diagonal elements are regarded as zero.

- `sparsematrix(m, n, g, `*`Diagonal`*`)` returns the sparse matrix whose i-th diagonal element is `g(i)`, where the index i runs from 1 to $\min(m, n)$. Cf. example 10.

Option <Banded>:

- With this option, sparse banded Toeplitz matrices can be created.

  A *banded sparse matrix* has zero entries outside the main diagonal and some of the adjacent sub- and superdiagonals.

- `sparsematrix(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 $2\,h + 1$, where h must not exceed `n`. The bandwidth of the resulting sparse matrix is at most h.

  All elements of the main diagonal of the created sparse matrix are initialized with the middle element of `List`. All elements of the i-th subdiagonal are initialized with the $(h + 1 - i)$-th element of `List`. All elements of the i-th superdiagonal are initialized with the $(h + 1 + i)$-th element of `List`. All entries on the remaining sub- and superdiagonals are regarded as zero.

  Cf. example 11.

Example 1. We create a 2×2 sparse matrix by passing a list of two rows to `sparsematrix`, where each row is a list of two elements:

```
>> A := sparsematrix([[1, 5], [2, 3]])

                          +-      -+
                          |  1, 5  |
                          |        |
                          |  2, 3  |
                          +-      -+
```

In the same way, we generate the following 2×3 sparse matrix:

```
>> B := sparsematrix([[-1, 5/2, 3], [1/3, 0, 2/5]])

                      +-                -+
                      |   -1, 5/2,  3   |
                      |                  |
                      |  1/3,  0,  2/5   |
                      +-                -+
```

We can do matrix arithmetic using the standard arithmetical operators of MuPAD. For example, the matrix product $A \cdot B$, the fourth power of A, and the scalar multiplication of A by $\frac{1}{3}$ are given by:

```
>> 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    |
+-                 -+  +-            -+  +-            -+
```

Since the dimensions of the sparse matrices A and B differ, the sum of A and B is not defined and MuPAD returns an error message:

```
>> A + B
```

```
Error: dimensions do not match [(Dom::SparseMatrix(Dom::Expres\
sionField()))::_plus]
```

To compute the inverse of A, enter:

```
>> 1/A
```

```
+-             -+
|  -3/7,  5/7   |
|               |
|   2/7, -1/7   |
+-             -+
```

If a sparse matrix is not invertible, the result of this operation is `FAIL`:

```
>> C := sparsematrix([[2, 0], [0, 0]])
```

```
+-      -+
|  2, 0  |
|        |
|  0, 0  |
+-      -+
```

```
>> C^(-1)
```

```
FAIL
```

```
>> delete A, B, C:
```

Example 2. In addition to standard matrix arithmetic, the library `linalg` offers numerous functions handling sparse matrices. For example, the function `linalg::rank` determines the rank of a sparse matrix:

```
>> A := sparsematrix([[1, 5], [2, 3]])
```

```
+-      -+
|  1, 5  |
|        |
|  2, 3  |
+-      -+
```

```
>> linalg::rank(A)
```

```
                                  2
```

The function `linalg::eigenvectors` computes the eigenvalues and the eigenvectors of `A`:

```
>> linalg::eigenvectors(A)
```

```
 -- --                -- +-              -+ -- --
 |  |                 |  |    1/2         |  |  |
 |  |                 |  |  11            |  |  |
 |  |    1/2          |  |  ----- - 1/2   |  |  |
 |  |  11    + 2, 1,  |  |    2           |  |  |,
 |  |                 |  |                |  |  |
 |  |                 |  |       1        |  |  |
 -- --                -- +-              -+ -- --

    --                -- +-                 -+ -- -- --
    |                 |  |      1/2          |  |  |  |
    |                 |  |    11             |  |  |  |
    |        1/2      |  |  - ----- - 1/2    |  |  |  |
    |  2 - 11   , 1,  |  |      2            |  |  |  |
    |                 |  |                   |  |  |  |
    |                 |  |        1          |  |  |  |
    --                -- +-                 -+ -- -- --
```

To determine the dimension of a sparse matrix use the function `linalg::matdim`:

```
>> linalg::matdim(A)
```

```
                               [2, 2]
```

The result is a list of two positive integers, the row and column number of the sparse matrix.

Use `info(linalg)` to obtain a list of available functions, or enter `?linalg` for details about this library.

```
>> delete A:
```

Example 3. Matrix entries can be accessed with the index operator `[ ]`:

```
>> A := sparsematrix([[1, 2, 3, 4], [2, 0, 4, 1], [-1, 0, 5, 2]])
```

```
                        +-           -+
                        |  1, 2, 3, 4  |
                        |              |
                        |  2, 0, 4, 1  |
                        |              |
                        | -1, 0, 5, 2  |
                        +-           -+
```

```
>> A[2, 1] * A[1, 2] - A[3, 1] * A[1, 3]
```

```
                               7
```

You can redefine a sparse matrix entry by assigning a value to it:

```
>> A[1, 2] := a^2: A

                     +-               -+
                     |        2        |
                     |   1, a , 3, 4   |
                     |                 |
                     |   2,  0, 4, 1   |
                     |                 |
                     |  -1,  0, 5, 2   |
                     +-               -+
```

The index operator can also be used to extract submatrices. The following call creates a copy of the submatrix of A comprising the second and the third row and the first three columns of A:

```
>> A[2..3, 1..3]

                       +-           -+
                       |   2, 0, 4   |
                       |             |
                       |  -1, 0, 5   |
                       +-           -+
```

The index operator does *not* allow to replace a submatrix of a given sparse matrix by another matrix. Use `linalg::substitute` to achieve this.

```
>> delete A:
```

Example 4. Some system functions can be applied to sparse matrices. For example, if you have a sparse matrix with symbolic entries and want to have all entries in expanded form, simply apply the function `expand`:

```
>> delete a, b:
   A := sparsematrix([
     [(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)  |
                 +-                             -+
```

```
>> expand(A)

          +-                            -+
          |             2    2   2    2  |
          |  - 2 a b + a   + b , a   + b  |
          |                              |
          |        2    2        2    2  |
          |       a  + b ,      a  - b   |
          +-                            -+
```

You can differentiate all sparse matrix components with respect to some indeterminate:

```
>> diff(A, a)

          +-                  -+
          |  2 a - 2 b, 2 a   |
          |                   |
          |     2 a,    2 a   |
          +-                  -+
```

The following command evaluates all sparse matrix components at a given point:

```
>> subs(A, a = 1, b = -1)

          +-       -+
          |  4, 2   |
          |         |
          |  2, 0   |
          +-       -+
```

Note that the function `subs` does not evaluate the result of the substitution. For example, we define the following matrix:

```
>> A := sparsematrix([[sin(x), x], [x, cos(x)]])

          +-                  -+
          |  sin(x),     x    |
          |                   |
          |     x,    cos(x)  |
          +-                  -+
```

Then we substitute $x = 0$ in each matrix component:

```
>> B := subs(A, x = 0)

          +-                  -+
          |  sin(0),     0    |
          |                   |
          |     0,    cos(0)  |
          +-                  -+
```

You see that the sparse matrix components are not evaluated completely. For example, if you enter sin(0) directly, it evaluates to zero.

The function eval can be used to evaluate the result of the function subs. However, eval does not operate on matrices directly, and you must use the function map to apply the function eval to each sparse matrix component:

```
>> map(B, eval)

                         +-      -+
                         |  0, 0  |
                         |        |
                         |  0, 1  |
                         +-      -+
```

The function zip can be applied to sparse matrices. The following call combines two matrices A and B by dividing each component of A by the corresponding component of B:

```
>> A := sparsematrix([[4, 2], [9, 3]]):
   B := sparsematrix([[2, 1], [3, -1]]):
   A, B, zip(A, B, `/`)

             +-      -+  +-       -+  +-       -+
             |  4, 2  |  |  2,  1  |  |  2,  2  |
             |        |, |         |, |         |
             |  9, 3  |  |  3, -1  |  |  3, -3  |
             +-      -+  +-       -+  +-       -+

>> delete A, B:
```

Example 5. A vector is either an $m \times 1$ sparse matrix (a column vector) or a $1 \times n$ sparse matrix (a row vector). To create a vector with sparsematrix, pass the dimension of the vector and a list of vector components as argument to sparsematrix:

```
>> row_vector := sparsematrix(1, 3, [1, 2, 3]);
   column_vector := sparsematrix(3, 1, [1, 2, 3])

                         +-       -+
                         | 1, 2, 3 |
                         +-       -+

                           +-   -+
                           |  1  |
                           |     |
                           |  2  |
                           |     |
                           |  3  |
                           +-   -+
```

If the only argument of sparsematrix is a non-nested list or a one-dimensional array, the result is a column vector:

```
>> sparsematrix([1, 2, 3])

                              +-   -+
                              |  1  |
                              |     |
                              |  2  |
                              |     |
                              |  3  |
                              +-   -+
```

For a row vector r, the calls r[1, i] and r[i] both return the i-th vector component of r. Similarly, for a column vector c, the calls c[i, 1] and c[i] both return the i-th vector component of c.

We extract the second component of the vectors defined above:

```
>> row_vector[2] = row_vector[1, 2],
   column_vector[2] = column_vector[2, 1]

                            2 = 2, 2 = 2
```

Use the function linalg::vecdim to determine the number of components of a vector:

```
>> linalg::vecdim(row_vector), linalg::vecdim(column_vector)

                                3, 3
```

The number of components of a vector can also be determined directly by the call nops(vector).

The dimension of a vector can be determined as described above in the case of sparse matrices:

```
>> linalg::matdim(row_vector),
   linalg::matdim(column_vector)

                            [1, 3], [3, 1]
```

See the linalg package for functions working with vectors, and the help page of norm for computing vector norms.

```
>> delete row_vector, column_vector:
```

Example 6. A vector is either an $m \times 1$ sparse matrix (a column vector) or a $1 \times n$ sparse matrix (a row vector). To create a vector with `sparsematrix`, one may also pass the dimension of the vector and a table of vector components as argument to `sparsematrix`:

```
>> delete v1, v2, t1, t2:
   t1 := table():
   t1[1,1] := 1:
   t1[1,2] := 2:
   t1[1,3] := 3:
   v1 := sparsematrix(1, 3, t1);
```

```
                         +-         -+
                         | 1, 2, 3 |
                         +-         -+
```

```
>> t2 := table():
   t2[1,1] := 1:
   t2[2,1] := 2:
   t2[3,1] := 3:
   v2 := sparsematrix(3, 1, t2);
```

```
                           +-   -+
                           |  1  |
                           |     |
                           |  2  |
                           |     |
                           |  3  |
                           +-   -+
```

All functions applied to the vectors in the previous example (see above) can can also be used on these vectors.

```
>> delete t1, t2, v1, v2:
```

Example 7. In the following examples, we illustrate various calls of `sparsematrix` as described above. We start by passing a nested list to `sparsematrix`, where each inner list corresponds to a row of the sparse matrix:

```
>> sparsematrix([[1, 2], [2]])
```

```
                         +-        -+
                         |  1, 2  |
                         |        |
                         |  2, 0  |
                         +-        -+
```

The number of rows of the created created sparse matrix is the number of inner lists, namely $m = 2$. The number of columns is determined by the maximal number of entries of an inner list. In the example above, the first list is the longest one, and hence $n = 2$. The second list has only one element and, therefore, the second entry in the second row of the returned sparse matrix was set to zero.

In the following call, we use the same nested list, but in addition pass two dimension parameters to create a 4×4 matrix:

```
>> sparsematrix(4, 4, [[1, 2], [2]])

                 +-            -+
                 |  1, 2, 0, 0  |
                 |              |
                 |  2, 0, 0, 0  |
                 |              |
                 |  0, 0, 0, 0  |
                 |              |
                 |  0, 0, 0, 0  |
                 +-            -+
```

In this case, the dimension of the sparse matrix is given by the dimension parameters. As before, missing entries in an inner list correspond to zero, and in addition missing rows are treated as zero rows.

Example 8. A one- or two-dimensional array of arithmetical expressions, such as:

```
>> a := array(1..3, 2..4,
     [[1, 1/3, 0], [-2, 3/5, 1/2], [-3/2, 0, -1]]
   )

               +-                -+
               |    1,  1/3,  0   |
               |                  |
               |   -2,  3/5, 1/2  |
               |                  |
               |  -3/2,  0,   -1  |
               +-                -+
```

can be converted into a matrix as follows:

```
>> A := sparsematrix(a)

               +-                -+
               |    1,  1/3,  0   |
               |                  |
               |   -2,  3/5, 1/2  |
               |                  |
               |  -3/2,  0,   -1  |
               +-                -+
```

Arrays serve, for example, as an efficient structured data type for programming. However, arrays do not have any algebraic meaning, and no mathematical operations are defined for them. If you convert an array into a matrix, you can use the full functionality defined for sparse matrices as described above. For example, let us compute the matrix $2A - A^2$ and the Frobenius norm of A:

```
>> 2*A - A^2, norm(A, Frobenius)

             +-                       -+
             |   5/3,   2/15,  -1/6   |   1/2      1/2
             |                         |  2    4037
             |  -1/20, 113/75,  6/5   |, ------------
             |                         |      30
             |    -3,    1/2,   -3    |
             +-                       -+
```

Note that an array may contain uninitialized entries:

```
>> b := array(1..4): b[1] := 2: b[4] := 0: b

                     +-                -+
                     | 2, ?[2], ?[3], 0 |
                     +-                -+
```

sparsematrix cannot handle arrays that have uninitialized entries, and responds with an error message:

```
>> sparsematrix(b)

 Error: unable to define matrix over Dom::ExpressionField() [(D\
 om::SparseMatrix(Dom::ExpressionField()))::new]
```

We initialize the remaining entries of the array b and convert it into a matrix, or more precisely, into a column vector:

```
>> b[2] := 0: b[3] := -1: sparsematrix(b)

                           +-    -+
                           |   2  |
                           |      |
                           |   0  |
                           |      |
                           |  -1  |
                           |      |
                           |   0  |
                           +-    -+

>> delete a, A, b:
```

Example 9. We show how to create a sparse matrix whose components are defined by a function of the row and the column index. The entry in the i-th row and the j-th column of a Hilbert matrix (see also `linalg::hilbert`) is $1/(i + j - 1)$. Thus the following command creates a 2×2 Hilbert matrix:

```
>> sparsematrix(2, 2, (i, j) -> 1/(i + j - 1))

                  +-           -+
                  |   1,  1/2   |
                  |             |
                  |  1/2, 1/3   |
                  +-           -+
```

The following two calls produce different results. In the first call, `x` is regarded as an unknown function, while it is a constant in the second call:

```
>> delete x:
   sparsematrix(2, 2, x), sparsematrix(2, 2, (i, j) -> x)

          +-                    -+  +-       -+
          |  x(1, 1), x(1, 2)   |  |  x, x   |
          |                     |, |         |
          |  x(2, 1), x(2, 2)   |  |  x, x   |
          +-                    -+  +-       -+
```

Example 10. Diagonal matrices can be created by passing the option *Diagonal* and a list of diagonal entries:

```
>> sparsematrix(3, 4, [1, 2, 3], Diagonal)

                 +-             -+
                 |  1, 0, 0, 0   |
                 |               |
                 |  0, 2, 0, 0   |
                 |               |
                 |  0, 0, 3, 0   |
                 +-             -+
```

One can generate the 3×3 identity matrix as follows:

```
>> sparsematrix::identity(3)

                   +-          -+
                   |  1, 0, 0   |
                   |            |
                   |  0, 1, 0   |
                   |            |
                   |  0, 0, 1   |
                   +-          -+
```

Here are alternative ways to create this matrix:

```
>> sparsematrix(3, 3, [1 $ 3], Diagonal)
```

```
+-         -+
|  1, 0, 0  |
|           |
|  0, 1, 0  |
|           |
|  0, 0, 1  |
+-         -+
```

Equivalently, you can use a function of one argument:

```
>> sparsematrix(3, 3, i -> 1, Diagonal)
```

```
+-         -+
|  1, 0, 0  |
|           |
|  0, 1, 0  |
|           |
|  0, 0, 1  |
+-         -+
```

Since the integer `1` also represents a constant function, the following shorter call creates the same matrix:

```
>> sparsematrix(3, 3, 1, Diagonal)
```

```
+-         -+
|  1, 0, 0  |
|           |
|  0, 1, 0  |
|           |
|  0, 0, 1  |
+-         -+
```

To demonstrate the use of tables for creating sparse matrices we can also create the identity matrix above by the lines:

```
>> t := table(): t[1, 1] := 1: t[2, 2] := 1: t[3, 3] := 1:
   sparsematrix(3, 3, t)
```

```
+-         -+
|  1, 0, 0  |
|           |
|  0, 1, 0  |
|           |
|  0, 0, 1  |
+-         -+
```

```
>> delete t:
```

Example 11. Banded Toeplitz matrices can be created with the option *Banded*. The following command creates a tri-diagonal matrix with constant bands:

```
>> sparsematrix(4, 4, [-1, 2, -1], Banded)
```

```
+-                 -+
|   2, -1,  0,  0   |
|                   |
|  -1,  2, -1,  0   |
|                   |
|   0, -1,  2, -1   |
|                   |
|   0,  0, -1,  2   |
+-                 -+
```

Example 12. Sparse matrices can also be created by using a `table`:

```
>> t := table():
   t[1, 2] := 12:
   t[3, 1] := 31:
   t[3, 2] := 32:
   t
```

```
table(
  (3, 2) = 32,
  (3, 1) = 31,
  (1, 2) = 12
)
```

The missing table entries correspond to empty matrix entries:

```
>> A := sparsematrix(4, 6, t)
```

```
+-                      -+
|   0, 12, 0, 0, 0, 0    |
|                        |
|   0,  0, 0, 0, 0, 0    |
|                        |
|  31, 32, 0, 0, 0, 0    |
|                        |
|   0,  0, 0, 0, 0, 0    |
+-                      -+
```

By using tables, one can easily create large sparse matrices without being forced to define all zero entries of the matrix. Note that this is a great advantage over using arrays where every component has to be initialized before.

```
>> delete t, A:
```

Example 13. The method "doprint" of Dom::SparseMatrix() prints only the non-zero components of a sparse matrix:

```
>> A := sparsematrix(4, 6):
   A[1, 2]:= 12: A[3, 1]:= 31: A[3, 2]:= 32:
   A::dom::doprint(A)

 Dom::SparseMatrix()(4, 6, [(1, 2) = 12, (3, 1) = 31,

    (3, 2) = 32])

>> delete A:
```

Changes:

- sparsematrix is a new function.

---

split – split an object

split(object, f) splits the object into a list of three objects. The first list entry is an object consisting of those operands of the input object that satisfy a criterion defined by the procedure f. The second list entry is built from the operands that violate the criterion. The third list entry is built from the operands for which it is unknown whether the criterion is satisfied.

Call(s):

- split(object, f <, p1, p2, ...>)

Parameters:

object	—	a list, a set, a table, an expression sequence, or an expression of type DOM_EXPR
f	—	a procedure returning a Boolean value
p1, p2, ...	—	any MuPAD objects accepted by f as additional parameters

Return Value: a list with three objects of the same type as the input object.

Overloadable by: object

Related Functions: map, op, select, zip

Details:

- The function `f` must return a value that can be evaluated to one of the Boolean values `TRUE`, `FALSE`, or `UNKNOWN`. It may either return one of these values directly, or it may return an equation or an inequality that can be simplified to one of these values by the function `bool`.
- The function `f` is applied to all operands `x` of the input object via the call `f(x, p1, p2, ...)`. Depending on the result `TRUE`, `FALSE`, or `UNKNOWN`, this operand is inserted into the first, the second, or the third output object, respectively.

  The output objects are of the same type as the input object, i.e., a list is split into three lists, a set into three sets, a table into three tables etc.
- If the input object is an expression sequence, then neither the input sequence nor the output (a list containing three sequences) are flattened.
- Also "atomic" objects such as numbers or identifiers can be passed to `split` as first argument. Such objects are handled like sequences with a single operand.
- `split` is a function of the system kernel.

Example 1. The following command checks which of the integers in the list are prime:

```
>> split([1, 2, 3, 4, 5, 6, 7, 8, 9, 10], isprime)

              [[2, 3, 5, 7], [1, 4, 6, 8, 9, 10], []]
```

The return value is a list of three lists. The first list contains the prime numbers, the second list contains all other numbers. The third list is empty, because for any number of the input list, it can be decided whether it is prime or not.

Example 2. With the optional arguments `p1, p2, ...` one can use functions that need more than one argument. For example, `contains` is a handy function to be used with `split`. The following call splits a list of sets into those sets that contain `x` and those that do not:

```
>> split([{a, x, b}, {a}, {x, 1}], contains, x)

                 [[{a, b, x}, {x, 1}], [{a}], []]
```

The elements of the returned list are of of type `DOM_LIST`, because the given expression was a list. If the given expression is of another type, e.g., `DOM_SET`, also the elements of the result are of type `DOM_SET`, too:

```
>> split({{a, x, b}, {a}, {x, 1}}, contains, x)

                 [{{x, 1}, {a, b, x}}, {{a}}, {}]
```

Example 3. We use the function `is` to split an expression sequence into subsequences. This function returns `UNKNOWN` if it cannot derive the queried property:

```
>> split((-2, -1, a, 0, b, 1, 2), is, Type::Positive)
```

```
[(1, 2), (-2, -1, 0), (a, b)]
```

Example 4. We split a table of people marked as male or female:

```
>> people := table("Tom" = "m", "Rita" = "f", "Joe" = "m"):
   [male, female, dummy] := split(people, has, "m"):
```

```
>> male
```

```
table(
  "Joe" = "m",
  "Tom" = "m"
)
```

```
>> female
```

```
table(
  "Rita" = "f"
)
```

```
>> dummy
```

```
table()
```

```
>> delete people, male, female, dummy:
```

`sqrt` – the square root function

`sqrt(z)` represents the square root of z.

Call(s):

- `sqrt(z)`

Parameters:

`z` — an arithmetical expression

Return Value: an arithmetical expression.

Overloadable by: `z`

Side Effects: When called with a floating point argument, the function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Related Functions: `_power`, `isqrt`, `numlib::issqr`

Details:

- $x =$ `sqrt`(z) represents the solution of $x^2 = z$ that has a nonnegative real part. In particular, it represents the positive root for real positive z. For real negative z, it represents the complex root with positive imaginary part.
- A floating point result is returned for floating point arguments. Note that the branch cut is chosen as the negative real semi-axis. The values returned by `sqrt` jump when crossing this cut. Cf. example 2.
- Certain simplifications of the argument may occur. In particular, positive integer factors are extracted from some symbolic products. Cf. example 3.
- Note that `sqrt`(x^2) cannot be simplified to x for all complex numbers (e.g., `sqrt`$(x^2) = -x$ for real $x < 0$). Cf. example 4.
- Mathematically, `sqrt(z)` coincides with `z^(1/2)` = `_power(z,1/2)`. However, `sqrt` provides more simplifications than `_power`. Cf. example 5.

Example 1. We demonstrate some calls with exact and symbolic input data:

```
>> sqrt(2), sqrt(4), sqrt(36*7), sqrt(127)

                          1/2         1/2      1/2
                         2   , 2, 6 7   , 127

>> sqrt(1/4), sqrt(1/2), sqrt(3/4), sqrt(25/36/7), sqrt(4/127)

                          1/2   1/2      1/2        1/2
                         2     3      5 7      2 127
                   1/2, ----, ----, ------, --------
                          2     2      42        127

>> sqrt(-4), sqrt(-1/2), sqrt(1 + I)

                                    1/2          1/2
                       2 I, 1/2 I 2   , (1 + I)

>> sqrt(x), sqrt(4*x^(4/7)), sqrt(4*x/3), sqrt(4*(x + I))
```

```
 1/2      2/7   / 4 x \1/2               1/2
x    , 2 x    , | --- |    , (4 x + 4 I)
                \  3  /
```

Example 2. Floating point values are computed for floating point arguments:

```
>> sqrt(1234.5), sqrt(-1234.5), sqrt(-2.0 + 3.0*I)

    35.13545218, 35.13545218 I, 0.8959774761 + 1.674149228 I
```

A jump occurs when crossing the negative real semi axis:

```
>> sqrt(-4.0), sqrt(-4.0 + I/10^100), sqrt(-4.0 - I/10^100)

            2.0 I, 2.5e-101 + 2.0 I, 2.5e-101 - 2.0 I
```

Example 3. The square root of symbolic products involving positive integer factors is simplified:

```
>> sqrt(20*x*y*z)

                           1/2  1/2
                  2 (x y z)    5
```

Example 4. Square roots of squares are not simplified, unless the argument is real and its sign is known:

```
>> sqrt(x^2*y^4)

                           2  4 1/2
                         (x  y )

>> assume(x > 0): sqrt(x^2*y^4)

                              4 1/2
                          x (y )

>> assume(x < 0): sqrt(x^2*y^4)

                               4 1/2
                         - x (y )
```

Example 5. `sqrt` provides more simplifications than the `_power` function:

```
>> sqrt(4*x), (4*x)^(1/2) = _power(4*x, 1/2)

                        1/2        1/2         1/2
                     2 x    , (4 x)    = (4 x)
```

`strmatch` – match a pattern in a character string

`strmatch(text, pattern)` checks whether the strings `text` and `pattern` coincide. The pattern may contain wildcards.

`strmatch(text, pattern, Index)` checks whether the text contains the pattern as a substring. If so, the position of the first occurrence of `pattern` is returned.

Call(s):

- `strmatch(text, pattern)`
- `strmatch(text, pattern, Index)`

Parameters:

`text, pattern` — character strings

Options:

Index — makes `strmatch` look for substrings in `text` coinciding with the pattern. If the pattern is not found, `FALSE` is returned. Otherwise, the location of the first occurrence is returned as a list of two integers.

Return Value: Without *Index*, either `TRUE` or `FALSE` is returned. With *Index*, a list of two nonnegative integers or `FALSE` is returned.

Overloadable by: `text, pattern`

Related Functions: `_concat`, `length`, `substring`, `stringlib::contains`, `stringlib::pos`

Details:

- The pattern may contain the wildcards `\?` and `\*`. The wildcard `\?` represents any single character or no character. The wildcard `\*` represents arbitrary (possibly empty) substrings.

- The string `text` must not contain wildcards.
- In MuPAD strings, the character \ is represented by \\. Cf. example 3.
- The library `stringlib` provides further functions for handling strings.
- `strmatch` is a function of the system kernel.

Option <`Index`>**:**

- `strmatch(text, pattern, `*Index*`)` checks whether `text` contains the string `pattern` as a substring. If so, a list `[i, j]` is returned. The integer `i` is the index of the first character of the matching substring, `j` is the index of the last character. I.e., `substring(text, i, j-i+1) = pattern`. Only the first occurrence of `pattern` inside `text` is found. If no match is found, `FALSE` is returned.
- Note that indexing of the characters in `text` starts with 0. NOTE
- If a wildcard is used in `pattern`, then the *largest* match is found. E.g., in the text `"XXabcbXX"`, the pattern `"a\*b"` matches the substring `"abcb"` rather than the substring `"ab"`.

Example 1. We do a simple comparison of strings:

```
>> s := "Hamburg": strmatch(s, "Hamburg")

                                 TRUE

>> strmatch(s, "Ham"), strmatch(s, "burg")

                             FALSE, FALSE

>> delete s:
```

Example 2. This example demonstrates wildcards. The wildcard \? represents a single character or no character:

```
>> strmatch("Mississippi", "Miss\?issip\?i")

                                 TRUE
```

The wildcard * represents any string including the empty string:

```
>> strmatch("Mississippi", "Mi\*i\*pp\*i\*")

                                 TRUE
```

In the following call, no match is found:

```
>> strmatch("Mississippi", "Mis\?i\*ppi\*i")

                        FALSE
```

Example 3. The character `?` is not a wildcard:

```
>> strmatch("Mississippi", "Miss?issip\?i")

                        FALSE
```

In MuPAD strings, the character \ is represented as \\: Consequently, \\ is regarded as a single character:

```
>> s := "a\\b": s[0], s[1], s[2]

                   "a", "\\", "b"

>> strmatch("Missi\\ssippi", "Missi\?ssippi")

                        TRUE

>> delete s:
```

Example 4. With the option *Index*, you can check whether a string contains another string. If so, the position of the substring in the source string is returned:

```
>> strmatch("cdxxcd", "xx", Index)

                       [2, 3]
```

Only the first occurrence of the pattern is found:

```
>> strmatch("cdxxcd", "cd", Index)

                       [0, 1]
```

The largest match is found:

```
>> strmatch("cdxxcxcd", "x\*x", Index)

                       [2, 5]

>> strmatch("cdxxcd", "xx\*", Index)

                       [2, 5]
```

```
>> strmatch("cdxxcd", "\*xx\*", Index)
```

[0, 5]

`subs` – substitute into an object

`subs(f, old = new)` returns a copy of the object `f` in which all operands matching `old` are replaced by the value `new`.

Call(s):

- `subs(f, old = new` <, *`Unsimplified`*>`)`
- `subs(f, old1 = new1, old2 = new2, ...` <, *`Unsimplified`*>`)`
- `subs(f, [old1 = new1, old2 = new2, ...]` <, *`Unsimplified`*>`)`
- `subs(f, {old1 = new1, old2 = new2, ...}` <, *`Unsimplified`*>`)`
- `subs(f, table(old1 = new1, old2 = new2, ...)` <, *`Unsimplified`*>`)`
- `subs(f, s1, s2, ...` <, *`Unsimplified`*>`)`

Parameters:

`f`	—	an arbitrary MuPAD object
`old, old1, old2, ...`	—	arbitrary MuPAD objects
`new, new1, new2, ...`	—	arbitrary MuPAD objects
`s1, s2, ...`	—	either equations `old = new`, or lists or sets of such equations, or tables whose entries are interpreted as such equations.

Options:

`Unsimplified`	—	prevents simplification of the returned object after substitution

Return Value: a copy of the input object with replaced operands.

Overloadable by: `f`

Related Functions: `extnops`, `extop`, `extsubsop`, `has`, `map`, `match`, `op`, `subsex`, `subsop`

Details:

- `subs` returns a modified copy of the object, but does not change the object itself.
- `subs(f, old = new)` searches `f` for operands matching `old`. Each such operand is replaced by `new`. Cf. example 1.
- The call `subs(f, old1 = new1, old2 = new2, ...)` invokes a “sequential substitution”: the specified substitutions are processed in sequence from left to right. Each substitution is carried out and the result is processed further with the next substitution. Cf. example 3.
- The call `subs(f, [old1 = new1, old2 = new2, ...])` invokes a “parallel substitution”; each substitution refers to the operands of the original input object `f`, not to the operands of “intermediate results” produced by previous substitutions. If multiple substitutions of an operand are specified, only the first one is carried out. Parallel substitution is also invoked when the substitutions are specified by sets or tables. Cf. example 4.
- The call `subs(f, s1, s2, ...)` describes the most general form of substitution which may combine sequential and parallel substitutions. This call is equivalent to `subs(... subs(subs(f, s1), s2), ...)`. Depending on the form of `s1`, `s2`, ..., sequential or parallel substitutions as described above are carried out in each step. Cf. example 5.
- Only operands accessible via the function `op` are replaced (“syntactical substitution”). A more “semantical” substitution is available with the function `subsex`, which also identifies and replaces partial sums and products. Cf. example 6.
- After substitution, the result is not evaluated. Use the function `eval` to enforce evaluation. Cf. example 7.
- Operands of expression sequences can be replaced by `subs`. Such objects are not flattened. Cf. example 8.
- The call `subs(f)` is allowed; it returns `f` without modifications.
- `subs` is a function of the system kernel.

Option <`Unsimplified`>:

- As the last step of a substitution, the modified object is simplified (however, not evaluated). This option suppresses this final simplification. Cf. example 9.

Example 1. We demonstrate some simple substitutions:

```
>> subs(a + b*a, a = 4)

                    4 b + 4

>> subs([a * (b + c), sin(b +c)], b + c = a)

                   2
                 [a , sin(a)]
```

Example 2. To replace the sine function in an expression, one has to prevent the evaluation of the identifier `sin` via `hold`. Otherwise, `sin` is replaced by its value, i.e., by the function environment defining the system's sine function. Inside the expression `sin(x)`, the 0-th operand `sin` is the identifier, not the function environment:

```
>> domtype(sin), domtype(hold(sin)), domtype(op(sin(x), 0));

        DOM_FUNC_ENV, DOM_IDENT, DOM_IDENT

>> subs(sin(x), sin = cos), subs(sin(x), hold(sin) = cos)

                sin(x), cos(x)
```

Example 3. The following call leads to a sequential substitution `x` $\to$ `y` $\to$ `z`:

```
>> subs(x^3 + y*z, x = y, y = z)

                     2    3
                    z  + z
```

Example 4. We demonstrate the difference between sequential and parallel substitutions. Sequential substitutions produce the following results:

```
>> subs(a^2 + b^3, a = b, b = a)

                     2    3
                    a  + a

>> subs(a^2 + b^3, b = a, a = b)

                     2    3
                    b  + b
```

In contrast to this, parallel substitution swaps the identifiers:

```
>> subs(a^2 + b^3, [a = b, b = a])

                         3    2
                        a  + b
```

In the following call, substitution of `y + x` for `a` yields the intermediate result `y + 2*x`. From there, substitution of `z` for `x` yields `y + 2 z`:

```
>> subs(a + x, a = x + y, x = z)

                        y + 2 z
```

Parallel substitution produces a different result. In the next call, `x + y` is substituted for `a`. Simultaneously, the operand `x` *of the original expression* `a + x` is replaced by `z`:

```
>> subs(a + x, [a = x + y, x = z])

                       x + y + z
```

The same happens when the substitutions are specified by a set of equations:

```
>> subs(a + x, {a = x + y, x = z})

                       x + y + z
```

Further, parallel substitution is used when specifying the substitutions by a table:

```
>> T := table(): T[a] := x + y: T[x] := z: T

                      table(
                        x = z,
                        a = x + y
                      )

>> subs(a + x, T)

                       x + y + z

>> delete T:
```

Example 5. We combine sequential and parallel substitutions:

```
>> subs(a + x, {a = x + y, x = z}, x = y)

                        2 y + z
```

Example 6. Only operands found by `op` are replaced. The following expression contains the subexpression `x + y` as the operand `op(f, [1, 2])`:

```
>> f := sin(z*(x + y)): op(f, [1, 2]);

                              x + y
```

Consequently, this subexpression can be replaced:

```
>> subs(f, x + y = z)

                                 2
                              sin(z )
```

Syntactically, the following sum does not contain the subexpression `x + y`. Consequently, it is not replaced by the following call:

```
>> subs(x + y + z, x + y = z)

                             x + y + z
```

In contrast to `subs`, the function `subsex` finds and replaces partial sums and products:

```
>> subsex(x + y + z, x + y = z)

                                2 z

>> subs(a*b*c, a*c = 5), subsex(a*b*c, a*c = 5)

                             a b c, 5 b

>> delete f:
```

Example 7. The result of `subs` is not evaluated. In the following call, the identifier `sin` is not replaced by its value, i.e., by the procedure defining the behavior of the system's sine function. Consequently, `sin(PI)` is not simplified to 0 by this procedure:

```
>> subs(sin(x), x = PI)

                              sin(PI)
```

The function `eval` enforces evaluation:

```
>> eval(subs(sin(x), x = PI))

                                 0
```

Example 8. Operands of expression sequences can be subtituted. Note that sequences need to be enclosed in brackets:

```
>> subs((a, b, a*b), a = x)

                                x, b, b x
```

Example 9. The option *Unsimplified* suppresses simplification:

```
>> subs(a + b + 2, a = 1, b = 0, Unsimplified)

                                1 + 0 + 2
```

Example 10. If we try to substitute something in a domain, the substitution is ignored. We define a new domain with the methods "foo" and "bar":

```
>> mydomain := newDomain("Test"):
   mydomain::foo := x -> 4*x:
   mydomain::bar := x -> 4*x^2:
```

Now we try to replace every 4 inside the domain by 3:

```
>> mydomain := subs(mydomain, 4 = 3):
```

However, this substitution did not have any effect:

```
>> mydomain::foo(x), mydomain::bar(x)

                                        2
                                4 x, 4 x
```

To substitute objects in a domain method, we have to substitute in the individual methods:

```
>> mydomain::foo := subs(mydomain::foo, 4 = 3):
   mydomain::bar := subs(mydomain::bar, 4 = 3):
   mydomain::foo(x), mydomain::bar(x)

                                        2
                                3 x, 3 x

>> delete mydomain:
```

`subsex` – extended substitution

`subsex(f, old = new)` returns a copy of the object `f` in which all expressions matching `old` are replaced by the value `new`. In contrast to the function `subs`, `subsex` also replaces “incomplete” subexpressions.

Call(s):

- `subsex(f, old = new` <, *Unsimplified*>`)`
- `subsex(f, old1 = new1, old2 = new2, ...` <, *Unsimplified*>`)`
- `subsex(f, [old1 = new1, old2 = new2, ...]` <, *Unsimplified*>`)`
- `subsex(f, {old1 = new1, old2 = new2, ...}` <, *Unsimplified*>`)`
- `subsex(f, table(old1 = new1, old2 = new2, ...)` <, *Unsimplified*>`)`
- `subsex(f, s1, s2, ...` <, *Unsimplified*>`)`

Parameters:

`f`	—	an arbitrary MuPAD object
`old, old1, old2, ...`	—	arbitrary MuPAD objects
`new, new1, new2, ...`	—	arbitrary MuPAD objects
`s1, s2, ...`	—	either equations `old = new`, or lists or sets of such equations, or tables whose entries are interpreted as such equations.

Options:

Unsimplified	—	prevents simplification of the returned object after substitution

Return Value: a copy of the input object with replaced operands.

Overloadable by: `f`

Related Functions: `extnops`, `extop`, `extsubsop`, `has`, `map`, `match`, `op`, `subs`, `subsop`

Details:

- `subsex` returns a modified copy of the object, but does not change the object itself.
- `subsex(f, old = new)` searches `f` for subexpressions matching `old`. Each such subexpression is replaced by `new`.
- In most cases, `subsex` leads to the same result as `subs`. However, in contrast to `subs`, `subsex` allows to replace "incomplete" subexpressions such as `a + b` in a sum `a + b + c`. In general, combinations of the operands of the n-ary "operators" `+`, `*`, `and`, `_exprseq`, `intersect`, `or`, `_lazy_and`, `_lazy_or`, and `union` can be replaced. In particular, partial sums and partial products can be replaced. Note that these operations are assumed to be commutative, e.g., `subsex(a*b*c, a*c = new)` does replace the partial product `a*c` by `new`. Cf. examples 1 and 2.

- `subsex` is much slower than `subs`! If `subs` can do the substitution, use `subs` rather than `subsex`.
- The call `subsex(f, old1 = new1, old2 = new2, ...)` invokes a "sequential substitution". See the `subs` help page for details.
- The call `subsex(f, [old1 = new1, old2 = new2, ...])` invokes a "parallel substitution". See the `subs` help page for details.
- The call `subsex(f, s1, s2, ...)` describes the most general form of substitution which may combine sequential and parallel substitutions. This call is equivalent to `subsex(... subsex(subsex(f, s1), s2), ...)`. Depending on the form of `s1`, `s2`, ..., sequential or parallel substitutions are carried out in each step. An example can be found on the `subs` help page.
- After substitution, the result is not evaluated. Use the function `eval` to enforce evaluation. Cf. example 3.
- Operands of expression sequences can be replaced by `subsex`. Such objects are not flattened. Cf. example 4.
- The call `subsex(f)` is allowed; it returns `f` without modifications.
- `subsex` is a function of the system kernel.

Option <`Unsimplified`>:

- As the last step of a substitution, the modified object is simplified (however, not evaluated). This option suppresses this final simplification. An example can be found on the `subs` help page.

Example 1. We demonstrate some simple substitutions; `subsex` finds and replaces partial sums and products:

```
>> subsex(a + b + c, a + c = x)

                              b + x

>> subsex(a*b*c, a*c = x)

                               b x

>> subsex(a * (b + c) + b + c, b + c = a)

                                 2
                             a + a

>> subsex(a + b*c*d + b*d, b*d = c);

                                   2
                           a + c + c
```

Example 2. We replace subexpressions inside an expression sequence and a symbolic union of sets:

```
>> subsex((a, b, c, d), (b, d) = w)

                     a, c, w

>> subsex(a union b union c, a union b = w)

                     c union w
```

The same can be achieved by using the functional equivalent `_union` of the operator `union`:

```
>> subsex(_union(a, b, c), _union(a, b) = w)

                     c union w
```

Example 3. The result of `subsex` is not evaluated. In the following call, the identifier `sin` is not replaced by its value, i.e., by the procedure defining the behavior of the system's sine function. Consequently, `sin(2*PI)` is not simplified to 0 by this procedure:

```
>> subsex(sin(2*x*y), x*y = PI)

                     sin(2 PI)
```

The function `eval` enforces evaluation:

```
>> eval(subsex(sin(2*x*y), x*y = PI))

                         0
```

Example 4. Operands of expression sequences can be subtituted. Note that sequences need to be enclosed in brackets:

```
>> subsex((a, b, a*b*c), a*b = x)

                     a, b, c x
```

Example 5. The option *Unsimplified* suppresses simplification:

```
>> subsex(2 + a + b, a + b = 0, Unsimplified)

                              2 + 0
```

`subsop` – replace operands

`subsop(object, i = new)` returns a copy of the object in which the `i`-th operand is replaced by the value `new`.

Call(s):

- `subsop(object, i1 = new1, i2 = new2, ...` <, *Unsimplified*>)

Parameters:

`object`	—	any MuPAD object
`i1, i2, ...`	—	integers or lists of integers
`new1, new2, ...`	—	arbitrary MuPAD objects

Options:

Unsimplified	—	prevents simplification of the returned object after substitution

Return Value: the input object with replaced operands or `FAIL`.

Overloadable by: `object`

Related Functions: `extnops`, `extop`, `extsubsop`, `map`, `match`, `op`, `subs`, `subsex`

Details:

- `subsop` returns a modified copy of the object, but does not change the object itself.
- `subsop(object, i = new)` replaces the operand `op(object, i)` by `new`. Operands are specified in the same way as with the function `op`: `i` may be an integer or a list of integers. E.g., `subsop(object, [j, k] = new)` replaces the suboperand `op(op(object, j), k)`. Cf. example 2.

  In contrast to `op`, ranges cannot be used in `subsop` to specify more than one operand to replace. Several substitution equations have to be specified instead.

- If several operands are to be replaced, the specified substitutions are processed in sequence from left to right. Each substitution is carried out and the result is processed further with the next substitution. The intermediate objects are not simplified.
- The result of `subsop` is not evaluated further. It can be evaluated via the function `eval`. Cf. example 3.
- Operands of expression sequences can be replaced by `subsop`. Such objects are not flattened.
- Note that the order of the operands may change by replacing operands and evaluating the result. Cf. example 4.
- `FAIL` is returned if an operand cannot be accessed.
- Substitution via `subsop` is faster than via `subs` or `subsex`.
- The call `subsop(object)` is allowed; it returns the object without modifications.
- `subsop` is a function of the system kernel.

Option <`Unsimplified`>**:**

- As the last step of a substitution, the modified object is simplified (however, not evaluated). This option suppresses this final simplification. Cf. example 3.

Example 1. We demonstrate how to replace one or more operands of an expression:

```
>> x := a + b: subsop(x, 2 = c)

                              a + c

>> subsop(x, 1 = 2, 2 = c)

                              c + 2
```

Also the 0-th operand of an expression (the "operator") can be replaced:

```
>> subsop(x, 0 = _mult)

                               a b
```

The variable `x` itself was not affected by the substitutions:

```
>> x

                              a + b

>> delete x:
```

Example 2. The following call specifies the suboperand `c` by a list of integers:

```
>> subsop([a, b, f(c)], [3, 1] = x)

                    [a, b, f(x)]
```

Example 3. This example demonstrates the effect of simplification. The following substitution replaces the first operand `a` by 2. The result simplifies to 3:

```
>> subsop(a + 1, 1 = 2)

                         3
```

The option *Unsimplified* suppresses the simplification:

```
>> subsop(a + 1, 1 = 2, Unsimplified)

                       2 + 1
```

The next call demonstrates the difference between *simplification* and *evaluation*. After substitution of `PI` for `x`, the identifier `sin` is not evaluated, i.e., the body of the system function `sin` is not executed:

```
>> subsop(sin(x),  1 = PI)

                      sin(PI)
```

Evaluation of `sin` simplifies the result:

```
>> eval(%)

                         0
```

Example 4. The order of operands may change by substitutions. Substituting `z` for the identifier `b` changes the internal order of the terms in `x`:

```
>> x := a + b + c: op(x)

                      a, b, c

>> x := subsop(x, 2 = z): op(x)

                      a, c, z

>> delete x:
```

Background:

- For overloading `subsop`, it is sufficient to handle the cases `subsop(object)` and `subsop(object, i = new)`.

  The case where the position of the operand to be replaced is given by a list is always handled recursively: First, `op` is called with the list bar the last element to find the object to substitute in (using the overloading of `op` if present, storing all the intermediate results), then the substitution is performed on that sub-object (using the overloading of `subsop` of the form `subsop(subobj, i = new)`). The result is substituted into the last-but-one result of the recursive `op` call, again respecting any overloading of `subsop`, and so on up to the front of the list.

`substring` – extract a substring from a string

`substring(string, i)` returns the $(i + 1)$-st character of a string.

`substring(string, i, l)` returns the substring of length `l` starting with the $(i + 1)$-st character of the string.

`substring(string, i..j)` returns the substring consisting of the characters $i + 1$ through $j + 1$.

Call(s):

- `substring(string, i)`
- `substring(string, i, l)`
- `substring(string, i..j)`

Parameters:

`string`	—	a nonempty character string
`i`	—	an integer between `0` and `length(string) - 1`
`l`	—	an integer between `0` and `length(string) - i`
`j`	—	an integer between `i` and `length(string) - 1`

Return Value: a character string.

Related Functions: `length`, `strmatch`, `stringlib::subs`

Details:

- The positions of the characters in a string are indexed from `0` to `length(string) - 1`. NOTE

- The empty string `""` is returned if the length `l` $= 0$ is specified.

⊞ `substring` is a function of the system kernel.

Example 1. We extract individual characters from a string:

```
>> substring("0123456789", i) $ i = 0..9

        "0", "1", "2", "3", "4", "5", "6", "7", "8", "9"
```

Substrings of various lengths are extracted:

```
>> substring("0123456789", 0, 2), substring("0123456789", 4, 4)

                          "01", "4567"
```

Substrings of length 0 are empty strings:

```
>> substring("0123456789", 4, 0)

                               ""
```

Ranges may be used to specify the substrings:

```
>> substring("0123456789", 0..9)

                          "0123456789"
```

Note that the position of the characters is counted from 0:

```
>> substring("123456789", 4..8)

                             "56789"
```

Example 2. The following `while` loop removes all trailing blank characters from a string:

```
>> string := "MuPAD       ":
   while substring(string, length(string) - 1) = " " do
     string := substring(string, 0..length(string) - 2)
   end_while

                             "MuPAD"
```

The following `for` loop looks for consecutive blank characters in a string and shrinks such spaces to a single blank character:

```
>> string := "MuPAD  -  the   open  computer     algebra system":
   result := substring(string, 0):
   space_count := 0:
   for i from 1 to length(string) - 1 do
     if substring(string, i) <> " " then
       result := result . substring(string, i);
       space_count := 0
     elif space_count = 0 then
       result := result . substring(string, i);
       space_count := space_count + 1
     end_if
   end_for:
   result

        "MuPAD - the open computer algebra system"

>> delete string, result, space_count, i:
```

---

sum – definite and indefinite summation

sum(f, i) computes a symbolic antidifference of $f(i)$ with respect to i.

sum(f, i = a..b) tries to find a closed form representation of the sum $\sum_{i=a}^{b} f(i)$.

Call(s):

- sum(f, i)
- sum(f, i = a..b)
- sum(f, i = RootOf(p, x))

Parameters:

f	—	an arithmetical expression depending on i
i	—	the summation index: an identifier
a, b	—	the boundaries: arithmetical expressions
p	—	a polynomial of type DOM_POLY or a polynomial expression
x	—	an indeterminate of p

Return Value: an arithmetical expression.

Related Functions: _plus, +, int, numeric::sum, product, rec

Details:

- `sum` serves for simplifying *symbolic* sums (the discrete analog of integration). It should *not* be used for adding a finite number of terms: if `a` and `b` are integers of type `DOM_INT`, the call `_plus(f $ i = a..b)` is more efficient than `sum(f, i = a..b)`. See example 3.

- `sum(f, i)` computes the indefinite sum of `f` with respect to `i`. This is an expression `g` such that $f(i) = g(i+1) - g(i)$.

- `sum(f, i = a..b)` computes the definite sum with `i` running from `a` to `b`.

  If `b - a` is a nonnegative integer, then the explicit sum $f(a) + f(a+1) + \cdots + f(b)$ is returned.

  If `b - a` is a negative integer, then the negative of the result of `sum(f, i = b+1..a-1)` is returned. With this convention, the rule

  `sum(f, i = a..b) + sum(f, i = b+1..c) = sum(f, i = a..c)`

  is satisfied for any `a`, `b`, and `c`.

- `sum(f, i = RootOf(p, x))` computes the sum with `i` extending over all roots of the polynomial `p` with respect to `x`.

  If `f` is a rational function of `i`, a closed form of the sum will be found.

  See example 2.

- The system returns a symbolic call of `sum` if it cannot compute a closed form representation of the sum.

- Infinite symbolic sums without symbolic parameters can be evaluated numerically via `float` or `numeric::sum`. Cf. example 4.

Example 1. We compute some indefinite sums:

```
>> sum(1/(i^2 - 1), i)

                          1       1
                      - --- - ---------
                        2 i   2 (i - 1)

>> sum(1/i/(i + 2)^2, i)

                psi(i + 2, 1)    1       1
                ------------- - --- - -------
                      2         4 i   4 i + 4

>> sum(binomial(n + i, i), i)
```

```
                 i binomial(i + n, i)
                 --------------------
                        n + 1
```

```
>> sum(binomial(n, i)/2^n - binomial(n + 1, i)/2^(n + 1), i)
```

```
       2 i binomial(n, i) - i binomial(n + 1, i)
       -----------------------------------------
                    n        n        n
                 2 2  - 4 i 2  + 2 n 2
```

We compute some definite sums. Note that $\pm\infty$ are valid boundaries:

```
>> sum(1/(i^2 + 21*i), i = 1..infinity)
```

```
                  18858053/108636528
```

```
>> sum(1/i, i = a .. a + 3)
```

```
               1     1       1       1
               - + ----- + ----- + -----
               a   a + 1   a + 2   a + 3
```

Example 2. We compute some sums over all roots of a polynomial:

```
>> sum(i^2, i = RootOf(x^3 + a*x^2 + b*x + c, x))
```

```
                        2
                       a  - 2 b
```

```
>> sum(1/(z + i), i = RootOf(x^4 - y*x + 1, x))
```

```
                              3
                       y + 4 z
                     ------------
                            4
                     y z + z  + 1
```

Example 3. `sum` can compute finite sums with integer boundaries of type `DOM_INT`:

```
>> sum(1/(i^2 + i), i = 1..100)
```

```
                        100/101
```

```
>> sum(binomial(n, i), i = 0..4)
```

```
    n + binomial(n, 2) + binomial(n, 3) + binomial(n, 4) + 1
```

```
>> expand(%)
```

```
                     2    3    4
               7 n   11 n    n    n
               --- + ----- - -- + -- + 1
                12     24    12   24
```

However, it is usually more efficient to use `_plus` in such a case:

```
>> _plus(1/(i^2 + i) $ i = 1..100)
```

```
                              100/101
```

```
>> _plus(binomial(n, i) $ i = 0..4)
```

```
    n + binomial(n, 2) + binomial(n, 3) + binomial(n, 4) + 1
```

However, if one of the boundaries is symbolic, then `_plus` cannot be used:

```
>> _plus(1/(i^2 + i) $ i = 1..n)
```

```
 Error: Illegal argument [_seqgen]
```

```
>> _plus(binomial(n, i) $ i = 0..n)
```

```
 Error: Illegal argument [_seqgen]
```

```
>> sum(1/(i^2 + i), i = 1..n), sum(binomial(n, i), i = 0..n)
```

```
                                n     n
                              -----, 2
                              n + 1
```

Example 4. The following infinite sum cannot be computed symbolically:

```
>> sum(ln(i)/i^5, i = 1..infinity)
```

```
                  / ln(i)                  \
               sum| -----, i = 1..infinity |
                  |    5                   |
                  \   i                    /
```

We obtain a floating point approximation via `float`:

```
>> float(%)
```

```
                            0.02857378051
```

Alternatively, the function `numeric::sum` can be used directly. This is usually much faster than applying `float`, since it avoids the overhead of `sum` attempting to compute a symbolic representation:

```
>> numeric::sum(ln(i)/i^5, i = 1..infinity)

                              0.02857378051
```

Example 5. `sum` does not find a closed form for the following definite sum. It returns a recurrence formula (see `rec`) for the sum instead:

```
>> sum(binomial(n, i)^3, i = 0..n)

      /    /                             2
      |    |              8 u2(n) (n + 1)
 solve| rec| u2(n + 2) - ---------------- -
      |    |                        2
      \    \                 (n + 2)

                          2
    u2(n + 1) (21 n + 7 n  + 16)
    ---------------------------, u2(n), {u2(0) = 1, u2(1) = 2}
                       2
              (n + 2)

    \ \
    | |
    | |
    | |
    / /
```

Background:

- The function `sum` implements Abramov's algorithm for rational expressions, Gosper's algorithm for hypergeometric expressions, and Zeilberger's algorithm for the definite summation of holonomic expressions.

`sysname` – the name of the operating system

`sysname()` returns information on the operating system on which MuPAD is currently executed.

Call(s):

⌘ `sysname(<Arch>)`

Options:

Arch — makes `sysname` return more specific information on the architecture

Return Value: a character string.

Related Functions: `system`

Details:

⌘ `sysname()` returns one of the following strings:

- `"UNIX"` for UNIX and Linux operating systems,
- `"MSDOS"` for MSDOS operating systems including MS-Windows,
- `"MACOS"` for Apple Macintosh operating systems.

⌘ `sysname(Arch)` returns a more specific name of the operating system as a character string. On some architectures, this information is not available and the same string is returned as by `sysname()`.

⌘ `sysname` is a function of the system kernel.

Example 1. On an MS-DOS or MS-Windows operating system (Microsoft), `sysname` returns the following values:

```
>> sysname(), sysname(Arch)

                    "MSDOS", "MSDOS"
```

Example 2. On a current Linux operating system such as Linux 2.0 using libc-6.0, `sysname` returns the following values:

```
>> sysname(), sysname(Arch)

                    "UNIX", "linux"
```

Example 3. On a Solaris operating system (SunOS, Sun Microsystems), `sysname` returns the following values:

```
>> sysname(), sysname(Arch)

                        "UNIX", "Solaris"
```

Example 4. On an Apple Macintosh operating system, `sysname` returns the following values:

```
>> sysname(), sysname(Arch)

                        "MACOS", "MACOS"
```

`sysorder` – compare objects according to the internal order

`sysorder(object1, object2)` returns `TRUE` if MuPAD's internal order of `object1` is less than or equal to the order of `object2`. Otherwise, `FALSE` is returned.

Call(s):

- `sysorder(object1, object2)`

Parameters:

`object1, object2` — arbitrary MuPAD objects

Return Value: `TRUE` or `FALSE`.

Related Functions: `_less`, `listlib::removeDupSorted`, `sort`

Details:

- A unique internal order exists for almost all objects that are created in a MuPAD session. `sysorder` compares two objects according to this internal order.

  NOTE

  The exceptions are sets and tables. They do not have a unique internal order. This implies that also objects containing sets or tables as (sub)operands do not have an unique internal order. Cf. example 3.

- One should not try and use the internal order to sort objects according to specific criteria. E.g., its does not necessarily reflect the natural ordering of numbers or strings. Further, the internal order may differ between different MuPAD versions.

  The only feature one may rely upon is its uniqueness. Cf. example 4.

- `sysorder` is a function of the system kernel.

Example 1. We give some examples how `sysorder` behaves in the current MuPAD version. For nonnegative integer numbers, the internal order is equal to the natural order. However, for rational numbers or negative integers, this is not true:

```
>> sysorder(3, 4) = bool(3 <= 4),
   sysorder(45, 33) = bool(45 <= 33),
   sysorder(0, 4) = bool(0 <= 4)

             TRUE = TRUE, FALSE = FALSE, TRUE = TRUE
```

```
>> sysorder(1/3, 1/4) <> bool(1/3 <= 1/4),
   sysorder(-4, 2) <> bool(-4 <= 2),
   sysorder(-4, -2) <>  bool(-4 <= -2)

           TRUE <> FALSE, FALSE <> TRUE, FALSE <> TRUE
```

Example 2. For character strings or names of identifiers, the internal order is not lexicographical:

```
>> sysorder("abc", "baa"), sysorder("abc", "bab"),
   sysorder("abc", "bac")

                      FALSE, FALSE, TRUE
```

```
>> sysorder(abc, baa), sysorder(abc, bab), sysorder(abc, bac)

                      FALSE, FALSE, TRUE
```

Example 3. There is no unique internal order for sets and tables:

```
>> sysorder({1, 2, 3}, {4, 5, 6}), sysorder({4, 5, 6}, {1, 2, 3})

                         FALSE, FALSE
```

```
>> sysorder(table("a" = 42), table("a" = 43)),
   sysorder(table("a" = 43), table("a" = 42))

                         FALSE, FALSE
```

Example 4. We give a simple application of `sysorder`. Suppose, we want to implement a function `f`, say, whose only known property is its skewness `f(-x) = -f(x)`. Expressions involving `f` should be simplified automatically, e.g., `f(x) + f(-x)` should yield zero for any argument `x`. To achieve this, we use `sysorder` to decide, whether a call `f(x)` should return `f(x)` or `-f(-x)`:

```
>> f := proc(x) begin
          if sysorder(x, -x) then
               return(-procname(-x))
          else return(procname(x))
          end_if;
        end_proc:
```

For numerical arguments, `f` prefers to rewrite itself with positive arguments:

```
>> f(-3), f(3), f(-4.5), f(4.5), f(-2/3), f(2/3)

          -f(3), f(3), -f(4.5), f(4.5), -f(2/3), f(2/3)
```

For other arguments, the result is difficult to predict:

```
>> f(x), f(-x), f(sqrt(2) + 1), f(-sqrt(2) - 1)

                                1/2             1/2
          -f(-x), f(-x), - f(- 2    - 1), f(- 2    - 1)
```

With this implementation, expressions involving `f` simplify automatically:

```
>> f(x) + f(-x) - f(3)*f(x) + f(-3)*f(-x) + sin(f(7)) + sin(f(-7))

                                0

>> delete f:
```

`system` – execute a command of the operating system

`system("command")` executes a command of the operating system or a program, respectively.

Call(s):

- `system("command")`
- `!command`

Parameters:

`"command"` — a command of the operating system or a name of a program as a MuPAD character string

Return Value: the "error code": an integer.

Related Functions: `sysname`

Details:

- `!command` is almost equivalent to `system("command")`; however, `!command` does not return any value to the MuPAD session.
- The syntax `!command` is allowed during interactive input only, not when reading MuPAD input from a file. "`!`" must be the first character on the input line.
- `system` is not available in all MuPAD versions. In particular, versions running under a Windows operating system do not support this function. If not available, a call to `system` results in the following error message:

  `Error: Function not available for this client [system].`
- `system("command")` sends the command to the operating system. E.g., this command may start another application program on the computer. The return value 0 indicates that the command was executed successfully. Otherwise, an integer error code is returned which depends on the operating system and the command.
- If the called command writes output to `stderr` on UNIX systems, the output will go to MuPAD's `stderr`. If `system` is called in XMuPAD, the output will be redirected to the shell which called XMuPAD.
- `system` is a function of the system kernel.

Example 1. On a UNIX or Linux system, the `date` command is executed. The command output is printed to the screen, the error code 0 for successful execution is returned to the MuPAD session:

```
>> errorcode := system("date"):

 Fri Sep 29 14:42:13 MEST 2000

>> errorcode

                                    0
```

Now the `date` command is called with the command line option `'+%m'` in order to display the current month only:

```
>> errorcode := system("date '+%m'"):

 09
```

Missing the prefix '+' in the command line option of date, date and therefore system returns an error code. Note that the error output goes to stderr:

```
>> system("date '%m'")

 date: invalid date '%m'

                                  1

>> delete errorcode:
```

Example 2. The output of a program started with the system command cannot be accessed in MuPAD directly, but it can be redirected into a file and then be read using the read or ftextinput command:

```
>> system("echo communication example > comm_file"):
   ftextinput("comm_file")

                     "communication example"

>> system("rm -f comm_file"):
```

---

table – **create a table**

table() creates a new empty table.

table(index1 = entry1, index2 = entry2, ...) creates a new table with the given indices and entries.

Call(s):

- table()
- table(index1 = entry1, index2 = entry2, ...)

Parameters:

index1, index2, ... — the indices: arbitrary MuPAD objects
entry1, entry2, ... — the corresponding entries: arbitrary MuPAD objects

Return Value: an object of type DOM_TABLE.

Related Functions: _assign, _index, array, assignElements, delete, DOM_ARRAY, DOM_LIST, DOM_TABLE, indexval

Details:

- In MuPAD, tables are the most flexible objects for storing data. In contrast to arrays or lists, arbitrary MuPAD objects can be used as indices. Indexed access to table entries is fast and nearly independent of the size of the table. Thus, tables are suitable containers for large data.
- For a table `T`, say, an indexed call `T[index]` returns the corresponding entry. If no such entry exists, the indexed expression `T[index]` is returned symbolically.
- An indexed assignment of the form `T[index] := entry` adds a new entry to an existing table `T` or overwrites an existing entry associated with the index.
- `table` is used for the explicit creation of a table. There also is the following mechanism for creating a table implicitly.

  If the value of an identifier `T`, say, is neither a table nor an array nor a list, then an indexed assignment `T[index] := entry` is equivalent to `T := table(index = entry)`. I.e., implicitly, a new table with one entry is created. Cf. example 2.

  If the value of `T` was either a table or an array or a list, then the indexed assignment only inserts a new entry without changing the type of `T` implicitly.
- Table entries can be deleted with the function `delete`. Cf. example 3.
- `table` is a function of the system kernel.

Example 1. The following call creates a table with two entries:

```
>> T := table(a = 13, c = 42)

                    table(
                      c = 42,
                      a = 13
                    )
```

The data may be accessed via indexed calls. Note the symbolic result for the index `b` which does not have a corresponding entry in the table:

```
>> T[a], T[b], T[c]

                    13, T[b], 42
```

Entries of a table may be changed via indexed assignments:

```
>> T[a] := T[a] + 10: T
```

```
table(
  c = 42,
  a = 23
)
```

Expression sequences may be used as indices or entries, respectively. Note, however, that they have to be enclosed in brackets when using them as input parameters for `table`:

```
>> T := table((a, b) = "hello", a + b = (50, 70))

table(
  a + b = (50, 70),
  (a, b) = "hello"
)

>> T[a + b]

50, 70
```

Indexed access does not require additional brackets:

```
>> T[a, b] := T[a, b]." world": T

table(
  a + b = (50, 70),
  (a, b) = "hello world"
)

>> delete T:
```

Example 2. Below, a new table is created implicitly by an idexed assigment using an identifier `T` without a value:

```
>> delete T: T[4] := 7: T

table(
  4 = 7
)

>> delete T:
```

Example 3. Use delete to delete entries:

```
>> T := table(a = 1, b = 2, (a, b) = (1, 2))

                    table(
                      (a, b) = (1, 2),
                      b = 2,
                      a = 1
                    )

>> delete T[b], T[a, b]: T

                          table(
                            a = 1
                          )

>> delete T:
```

---

taylor – compute a Taylor series expansion

taylor(f, x = x0) computes the first terms of the Taylor series of f with respect to the variable x around the point x0.

Call(s):

taylor(f, x < = x0> <, order>)

Parameters:

f	—	an arithmetical expression representing a function in x
x	—	an identifier
x0	—	the expansion point: an arithmetical expression; if not specified, the default expansion point 0 is used.
order	—	the number of terms to be computed: a nonnegative integer; the default order is given by the environment variable ORDER (default value 6).

Return Value: an object of domain type Series::Puiseux or a symbolic expression of type "taylor".

Side Effects: The function is sensitive to the environment variable ORDER, which determines the default number of terms in series computations.

Overloadable by: f

Related Functions: asympt, diff, limit, O, series, Series::Puiseux, Type::Series

Details:

- `taylor` tries to compute the Taylor series of `f` around `x = x0`. Three cases can occur:

  1. `taylor` is able to compute the corresponding Taylor series. In this case, the result is a series expansion of domain type `Series::Puiseux`. Use `expr` to convert it to an arithmetical expression of domain type `DOM_EXPR`. Cf. example 1.

  2. `taylor` is able to decide that the corresponding Taylor series does not exist. In this case, an error is raised. Cf. example 2.

  3. `taylor` is not able to determine whether the corresponding Taylor series exists or not. Internally, the function `series` is called; it returns a symbolical call. In this case, also `taylor` returns a symbolic expression of type `"taylor"`. Cf. example 3.

- Mathematically, the expansion computed by `taylor` is valid in some open disc around the expansion point in the complex plane.

- If `x0` is `complexInfinity`, then an expansion around the complex infinity, i.e., the north pole of the Riemann sphere, is computed. If `x0` is `infinity` or `-infinity`, a directed series expansion valid along the real axis is computed.

  Such an expansion is computed as follows: The series variable `x` in `f` is replaced by $x = \pm 1/u$. Then a directed series expansion at $u = 0$ from the right is computed. If `x0 = complexInfinity`, then an undirected expansion around $u = 0$ is computed. Finally, $u = \pm 1/x$ is substituted in the result.

  Mathematically, the result of an expansion around `complexInfinity` or $\pm$`infinity` is a power series in $1/x$. Cf. example 4.

- The number of requested terms for the expansion is `order` if specified. Otherwise, the value of the environment variable `ORDER` is used. You can change the default value 6 by assigning a new value to `ORDER`.

  The number of terms is counted from the lowest degree term on for finite expansion points, and from the highest degree term on for expansions around infinity, i.e., "`order`" has to be regarded as a "relative truncation order".

  NOTE

  Note, however, that the actual number of terms in the resulting series expansion may differ from the requested number of terms. See the help page of `series` for details and examples.

- `taylor` uses the more general series function `series` to compute the Taylor expansion. See the corresponding help page for `series` for details about the parameters and the data structure of a Taylor series expansion.

Example 1. We compute a Taylor series around the default point 0:

```
>> s := taylor(exp(x^2), x)

                        4    6
                   2   x    x        8
              1 + x  + -- + -- + O(x )
                        2    6
```

The result of `taylor` is of the following domain type:

```
>> domtype(s)

                        Series::Puiseux
```

If we apply the function `expr` to a series, we get an arithmetical expression without the order term:

```
>> expr(s); domtype(%)

                        4    6
                   2   x    x
                  x  + -- + -- + 1
                        2    6

                        DOM_EXPR

>> delete s:
```

Example 2. A Taylor series expansion of $f(x) = \frac{1}{x^2-1}$ around $x = 1$ does not exist. Therefore, `taylor` responds with an error message:

```
>> taylor(1/(x^2 - 1), x = 1)

 Error: does not have a Taylor series expansion, try 'series' [\
 taylor]
```

Following the advice given in this error message, we try `series` to compute a more general series expansion. A Laurent expansion does exist:

```
>> series(1/(x^2 - 1), x = 1)

                                         2          3          4
     1              (x - 1)    (x - 1)    (x - 1)    (x - 1)
 --------- - 1/4 + ------- - -------- + -------- - -------- +
 2 (x - 1)             8         16         32         64

              5
    O((x - 1) )
```

Example 3. If a Taylor series expansion cannot be computed, then the function call with evaluated arguments is returned symbolically together with a warning:

```
>> taylor(1/exp(x^a), x = 0)

 Warning: could not compute Taylor series expansion; try 'serie\
 s' with option 'Left', 'Right', or 'Real' for a more general e\
 xpansion [taylor]

                        /    1          \
                  taylor| -------, x = 0 |
                        |      a         |
                        \ exp(x )        /
```

In this example, also `series` returns a symbolic function call. Even if you try one of the proposed options, `series` is not able to compute a series expansion.

Here is another example where no Taylor expansion can be computed. However, `series` with an optional argument yields a more general type of expansion in this case:

```
>> taylor(psi(1/x), x = 0)

 Warning: could not compute Taylor series expansion; try 'serie\
 s' with option 'Left', 'Right', or 'Real' for a more general e\
 xpansion [taylor]

                        /    / 1 \        \
                  taylor| psi| - |, x = 0 |
                        \    \ x /        /

>> series(psi(1/x), x = 0, Right)

                              2     4
                         x    x     x       6
                - ln(x) - - - -- + --- + O(x )
                         2   12   120


```

Example 4. This is an example of a directed Taylor expansion along the real axis around `infinity`:

```
>> taylor(exp(1/x), x = infinity)

              1     1      1      1        1      /  1 \
          1 + - + ---- + ---- + ----- + ------ + O| -- |
              x      2      3      4        5     |  6 |
                  2 x    6 x    24 x    120 x     \ x  /
```

In fact, this is even an undirected expansion:

```
>> taylor(exp(1/x), x = complexInfinity)

           1     1      1      1        1      /  1 \
       1 + - + ---- + ---- + ----- + ------ + O| -- |
           x      2      3       4        5    |  6 |
               2 x    6 x    24 x    120 x     \ x  /
```

Changes:

- Taylor expansions around `complexInfinity` are now possible.

`tbl2text` – concatenate the strings in a table

`tbl2text` concatenates all entries of a table of character strings.

Call(s):

- `tbl2text(strtab)`

Parameters:

`strtab` — a table of character strings

Return Value: a character string.

Related Functions: `_concat`, `coerce`, `expr2text`, `int2text`, `text2expr`, `text2list`, `text2tbl`

Details:

- The table must be indexed by `1`, `2`, `3` etc. All entries must be character strings. They are concatenated in the order of their indices.
- `tbl2text` restores strings split by `text2tbl`.
- `tbl2text` is a function of the system kernel.

Example 1. A character string can be created from an arbitrary number of table entries:

```
>> tbl2text(table(1 = "Hell", 2 = "o", 3 = " ", 4 = "world."))

                          "Hello world."
```

`tcoeff` – the trailing coefficient of a polynomial

`tcoeff(p)` returns the trailing coefficient of the polynomial `p`.

Call(s):

- `tcoeff(p <, vars> <, order>)`

Parameters:

`p`	—	a polynomial of type `DOM_POLY` or a polynomial expression
`vars`	—	a list of indeterminates of the polynomial: typically, identifiers or indexed identifiers
`order`	—	the term ordering: either *`LexOrder`*, or *`DegreeOrder`*, or *`DegInvLexOrder`*, or a user-defined term ordering of type `Dom::MonomOrdering`. The default is the lexicographical ordering *`LexOrder`*.

Return Value: an element of the coefficient domain of the polynomial or `FAIL`.

Overloadable by: `p`

Related Functions: `coeff`, `collect`, `degree`, `degreevec`, `ground`, `lcoeff`, `ldegree`, `lmonomial`, `lterm`, `nterms`, `nthcoeff`, `nthmonomial`, `nthterm`, `poly`, `poly2list`

Details:

- The argument `p` can either be a polynomial expression, or a polynomial generated by `poly`, or an element of some polynomial domain overloading `tcoeff`.
- If a list of indeterminates is provided, then `p` is regarded as a polynomial in these indeterminates. Note that the specified list does not have to coincide with the indeterminates of the input polynomial. Cf. example 1.

- The returned coefficient is "trailing" with respect to the lexicographical ordering, unless a different ordering is specified via the argument `order`. Cf. example 2.
- The result of `tcoeff` is not fully evaluated. Evaluation can be enforced by the function `eval`. Cf. example 3.
- `tcoeff` returns `FAIL` if the input polynomial cannot be converted to a polynomial in the specified indeterminates. Cf. example 4.
- With the ordering *LexOrder*, `tcoeff` calls a fast kernel function. Other orderings are handled by slower library functions.

Example 1. We demonstrate how the indeterminates influence the result:

```
>> p := 2*x^2*y + 3*x*y^2:
   tcoeff(p), tcoeff(p, [x, y]), tcoeff(p, [y, x])

                         2, 3, 2
```

Note that the indeterminates passed to `tcoeff` will be used, even if the polynomial provides different indeterminates :

```
>> p := poly(2*x^2*y + 3*x*y^2, [x, y]):
   tcoeff(p), tcoeff(p, [x, y]), tcoeff(p, [y, x]),
   tcoeff(p, [y]), tcoeff(p, [z])

                         2     2           2
               3, 3, 2, 2 x , 2 x  y + 3 x y

>> delete p:
```

Example 2. We demonstrate how various orderings influence the result:

```
>> p := poly(5*x^4 + 4*x^3*y + 3*x^2*y^3*z, [x, y, z]):
   tcoeff(p), tcoeff(p, DegreeOrder), tcoeff(p, DegInvLexOrder)

                         3, 4, 5
```

The following call uses the reverse lexicographical order on 3 indeterminates:

```
>> tcoeff(p, Dom::MonomOrdering(RevLex(3)))

                            5

>> delete p:
```

Example 3. The result of `tcoeff` is not fully evaluated:

```
>> p := poly(27*x^2 + a*x, [x]): a := 5:
   tcoeff(p, [x]), eval(tcoeff(p, [x]))

                          a, 5

>> delete p, a:
```

Example 4. We define a polynomial over the integers modulo 7:

```
>> p := poly(3*x, [x], Dom::IntegerMod(7)): tcoeff(p)

                          3 mod 7
```

This polynomial cannot be regarded as a polynomial with respect to another indeterminate, because the "coefficient" `3*x` cannot be interpreted as an element of the coefficient ring `Dom::IntegerMod(7)`:

```
>> tcoeff(p, [y])

                          FAIL

>> delete p:
```

`testargs` – decide whether procedure arguments should be tested

Inside a procedure, `testargs` indicates whether the procedure was called on the interactive level.

Call(s):

- `testargs()`
- `testargs(b)`

Parameters:

`b` — `TRUE` or `FALSE`

Return Value: `TRUE` or `FALSE`.

Related Functions: `proc`, `testtype`, `Pref::typeCheck`

Details:

- Checking the input parameters of a procedure may be costly. For this reason, most functions of the MuPAD libraries are implemented according to the following philosophy:

  If a procedure is called on the interactive level, i.e., if its parameters are supplied interactively by the user, then the parameters should be checked. If the input parameters do not comply with the documented specification of the procedure, then appropriate error messages should be returned to notify the user of wrong usage.

  If the procedure is called by another procedure, then no check of the parameters should be performed to improve efficiency. The calling procedure is supposed to make sure that appropriate parameters are passed.

  `testargs` is the tool to check whether the arguments should be tested: called inside the body of a procedure, `testargs()` returns `TRUE` if the procedure was called on the interactive level. Otherwise, it returns `FALSE`.

- `testargs` has two modes. In the "standard mode", its functionality is as described above. In the "debugging mode", the call `testargs()` always returns `TRUE`. This supports the debugging of procedures: any function using `testargs` checks its parameters and returns useful error messages if called in an inappropriate way.

  The call `testargs(TRUE)` switches to the "debugging mode", i.e., parameter testing is switched on globally.

  The call `testargs(FALSE)` switches to the "standard mode", i.e., parameter testing is used only on the interactive level.

  The call `testargs(b)` returns the previously set value.

- Checking the input parameters of a procedure can also be controlled with the function `Pref::typeCheck`.

- `testargs` is a function of the system kernel.

Example 1. The following example demonstrates how `testargs` should be used inside a procedure. The function `p` is to generate a sequence of `n` zeroes; its argument should be a positive integer:

```
>> p := proc(n)
   begin
     if testargs() then
        if not testtype(n, Type::PosInt) then
          error("expecting a positive integer");
        end_if;
     end_if;
     return(0 $ n)
   end_proc:
```

Its argument is checked when p is called on the interactive level:

```
>> p(13/2)

 Error: expecting a positive integer [p]
```

Calling p from within a procedure with an inappropriate parameter does not invoke the argument testing. The following error message is not issued by p. It is caused by the attempt to evaluate `0 $ n`:

```
>> f := proc(n) begin p(n) end_proc: f(13/2)

 Error: Illegal argument [_seqgen];
 during evaluation of 'p'
```

We switch on the "debugging mode" of testargs:

```
>> testargs(TRUE):
```

Now also a non-interactive call to p produces an informative error message:

```
>> f(13/2)

 Error: expecting a positive integer [p]
```

We clean up, restoring the "standard mode" of testargs:

```
>> testargs(FALSE): delete f, g:
```

---

testtype – syntactical type checking

`testtype(object, T)` checks whether the object is syntactically of type T.

Call(s):

- `testtype(object, T)`

Parameters:

object — any MuPAD object
T — a type object

Return Value: TRUE or FALSE.

Overloadable by: object, T

Related Functions: coerce, domtype, hastype, is, type, Type

Details:

- The type object `T` may be either a domain type such as `DOM_INT`, `DOM_EXPR` etc., a string as returned by the function `type`, or a `Type` object. The latter are probably the most useful predefined values for the argument `T`.
- `testtype` performs a purely syntactical check. Use `is` for semantical checks taking into account properties of identifiers!
- See the "background" section below for details on the overloading mechanism.
- `testtype` is a function of the system kernel.

Example 1. The following call tests, whether the first argument is an expression. Expressions are basic objects of domain type `DOM_EXPR`:

```
>> testtype(x + y, DOM_EXPR)

                    TRUE
```

The `type` function distinguishes expressions. The corresponding type string is a valid type object for `testtype`:

```
>> type(x + y), testtype(x + y, "_plus")

                "_plus", TRUE
```

The following call tests, whether the first argument is an integer by querying, whether it is of domain type `DOM_INT`:

```
>> testtype(7, DOM_INT)

                    TRUE
```

Note that `testtype` performs a purely syntactical test. Mathematically, the integer 7 is a rational number. However, the domain type `DOM_RAT` does not encompass `DOM_INT`:

```
>> testtype(7, DOM_RAT)

                    FALSE
```

The `Type` library provides more flexible type objects. E.g., `Type::Rational` represents the union of `DOM_INT` and `DOM_RAT`:

```
>> testtype(7, Type::Rational)

                    TRUE
```

The number 7 matches other types as well:

```
>> testtype(7, Type::PosInt), testtype(7, Type::Prime),
   testtype(7, Type::Numeric), testtype(7, Type::Odd)

                    TRUE, TRUE, TRUE, TRUE
```

Example 2. Subtypes of expressions can be specified via character strings:

```
>> type(f(x)),  type(sin(x))

                       "function", "sin"

>> testtype(sin(x), "function"), testtype(sin(x), "sin"),
   testtype(sin(x), "cos")

                       TRUE, TRUE, FALSE
```

Example 3. We demonstrate how to implement a customized type object "div3" which is to represent integer multiples of 3. One has to create a new domain with a "testtype" attribute:

```
>> div3 := newDomain("divisible by 3?"):
   div3 := slot(div3, "testtype",
                proc(x) begin
                    return(testtype(x/3, Type::Integer))
                end_proc):
```

Via overloading, the command `testtype(object, div3)` calls this slot:

```
>> testtype(5, div3), testtype(6, div3), testtype(sin(1), div3)

                      FALSE, TRUE, FALSE

>> delete div3:
```

Background:

- Overloading of `testtype` works as follows: First, it is checked whether `domtype(object) = T` or `type(object) = T` holds. If so, `testtype` returns `TRUE`.
- Next, the method `"testtype"` of the domain `object::dom` is called with the arguments `object, T`. If this method returns a result other than `FAIL`, then `testtype` returns this value.

- If the method `object::dom::testtype` does not exist or if this method returns `FAIL`, then overloading by the second argument is used:

  - If `T` is a domain, then the method `"testtype"` of `T` is called with the arguments `object, T`.
  - If `T` is not a domain, then the method `"testtype"` of `T::dom` is called with the arguments `object, T`.

`text2expr` – convert a character string to an expression

`text2expr(text)` interprets the character string `text` as MuPAD input and generates the corresponding object.

Call(s):

- `text2expr(text)`

Parameters:

`text` — a character string

Return Value: a MuPAD object.

Related Functions: `coerce`, `expr2text`, `input`, `int2text`, `tbl2text`, `text2int`, `text2list`, `text2tbl`

Details:

- The text must correspond to syntactically correct MuPAD input. Otherwise, `text2expr` produces an error. Typically, strings created from MuPAD objects via `expr2text` can be reconverted to corresponding objects.
- The object is returned without being further evaluated. Evaluation can be enforced using the function `eval`.
- The `text` does not need to be terminated with a "`;`" or a "`:`" character, respectively.
- `text2expr` is a function of the system kernel.

Example 1. A character string is converted to a simple expression. The newly created expression is not evaluated automatically:

```
>> text2expr("21 + 21")
```

```
21 + 21
```

It may be evaluated via `eval`:

```
>> eval(%)
```

```
42
```

Example 2. A character string is converted to a statement sequence:

```
>> text2expr("x:= 3; x + 2 + 1"); eval(%)
```

```
(x := 3;
x + 2 + 1)

6
```

```
>> x
```

```
3
```

```
>> delete x:
```

Example 3. A matrix is converted to a string:

```
>> matrix([[a11, a12], [a21, a22]])
```

```
+-          -+
|  a11, a12  |
|            |
|  a21, a22  |
+-          -+
```

```
>> expr2text(%)
```

```
"Dom::Matrix()(array(1..2, 1..2, (1,1) = a11, (1,2) = a12, (2,\
1) = a21, (2,2) = a22))"
```

The string is reconverted to a matrix:

```
>> text2expr(%)
```

```
 Dom::Matrix()(array(1..2, 1..2, (1, 1) = a11, (1, 2) = a12,

    (2, 1) = a21, (2, 2) = a22))
```

```
>> eval(%)
```

```
                              +-          -+
                              |  a11, a12  |
                              |            |
                              |  a21, a22  |
                              +-          -+
```

`text2int` – convert a character string to an integer

`text2int(text, b)` converts a character string corresponding to an integer in `b`-adic representation to an integer of type `DOM_INT`.

Call(s):

- `text2int(text <, b>)`

Parameters:

`text`	—	a character string
`b`	—	the base: an integer between 2 and 36. The default base is 10.

Return Value: an integer.

Related Functions: `coerce`, `expr2text`, `genpoly`, `int2text`, `numlib::g_adic`, `tbl2text`, `text2expr`, `text2list`, `text2tbl`

Details:

- The text is interpreted as a `b`-adic representation. Its must consist of the first `b` characters in $0, 1, \ldots, 9, A, B, \ldots, Z$. Also lower case letters $a, b, \ldots, z$ are accepted. For bases larger than 10, the letters can be used to represent the `b`-adic digits larger than 9: $a = A = 10, \ldots, z = Z = 35$.
- `text2int` is the inverse of `int2text`.
- `text2int` is a function of the system kernel.

Example 1. Relative to the default base 10, `text2int` provides a mere data-type conversion from `DOM_STRING` to `DOM_INT`:

```
>> text2int("123"), text2int("-45678")

                          123, -45678
```

Example 2. The characters of the input string are interpreted as digits with respect to the specified base, the return value is a standard MuPAD integer represented with respect to the decimal system. The following example converts integers from the base 2 and 16, respectively, to the base 10:

```
>> text2int("101", 2), text2int("101", 16)

                             5, 257
```

The digit "3" does not exist in a binary representation:

```
>> text2int("103", 2)

 Error: Illegal argument [text2int]
```

Example 3. For bases larger than 10, letters represent the `b`-adic digits larger than 9:

```
>> text2int("3B9ACA00", 16), text2int("Z", 36) = text2int("z", 36)

                       1000000000, 35 = 35
```

`text2list`, `text2tbl` – split a character string into substrings

`text2list` splits a character string into a list of substrings.

`text2tbl` splits a character string into a table of substrings.

Call(s):

- `text2list(text, separators` <, *Cyclic*>)
- `text2tbl(text, separators` <, *Cyclic*>)

Parameters:

`text`	—	the text to be analyzed: a character string
`separators`	—	delimiters: a list of character strings. The empty string `""` is not accepted as a delimiter.

Options:

Cyclic — the delimiter list is used cyclicly

Return Value: a list, respectively a table, of character strings.

Related Functions: `coerce`, `expr2text`, `int2text`, `tbl2text`, `text2expr`, `text2int`

Details:

- Both functions split a string into substrings, using the strings in the list `separators` as delimiters. `text2list` returns a list containing the substrings; `text2tbl` returns a table, using the indices `1`, `2`, `3` etc.

- Without the option *Cyclic*, the text is split as follows. The first occurrence of one of the delimiters in `separators` is located in `text`. If no delimiter is found, the full text is returned as the only substring. Otherwise, the substring up to the delimiter defines the first substring. The delimiter is the second substring. The remaining text is processed as above until there are no more characters left.

  Without *Cyclic*, the result does not depend on the order of the delimiters.

- With the option *Cyclic*, the first delimiter in `separators` is used to identify the first substring. The delimiter itself is the second substring. Then the second delimiter in `separators` is used to identify the third substring etc.

  After using the last delimiter of the list, the first one is used again, until the whole text is processed or until the current delimiter is not found in the remaining text.

  With *Cyclic*, the result depends on the order of the delimiters.

- `tbl2text` restores strings split by `text2tbl`.

- `text2list, text2tbl` are functions of the system kernel.

Example 1. The following example demonstrates the difference in calling `text2list` with and without the option *Cyclic*:

```
>> text2list("This is a simple example!", ["is", "mp"])

  ["Th", "is", " ", "is", " a si", "mp", "le exa", "mp", "le!"]

>> text2list("This is a simple example!", ["is", "mp"], Cyclic)

          ["Th", "is", " is a si", "mp", "le example!"]
```

Example 2. The following example demonstrates the difference in calling `text2tbl` with and without the option *Cyclic*:

```
>> text2tbl("This is a simple example!", ["is", "mp"])
```

```
table(
  9 = "le!",
  8 = "mp",
  7 = "le exa",
  6 = "mp",
  5 = " a si",
  4 = "is",
  3 = " ",
  2 = "is",
  1 = "Th"
)
```

```
>> text2tbl("This is a simple example!", ["is", "mp"], Cyclic)
```

```
table(
  5 = "le example!",
  4 = "mp",
  3 = " is a si",
  2 = "is",
  1 = "Th"
)
```

`textinput` – interactive input of text

`textinput` allows interactive input of text.

Call(s):

- `textinput(<prompt1>)`
- `textinput(<prompt1,> x1, <prompt2,> x2, ...)`

Parameters:

`prompt1, prompt2, ...`	—	input prompts: character strings
`x1, x2, ...`	—	identifiers

Return Value: the last input, converted to a character string.

Related Functions: `finput`, `fname`, `fprint`, `fread`, `ftextinput`, `input`, `print`, `read`, `text2expr`, `write`

Details:

- `textinput()` displays the prompt "`Please enter text :`" and waits for input by the user. The input is converted to a character string, which is returned as the function's return value.

- `textinput(prompt1)` uses the character string `prompt1` instead of the default prompt "`Please enter text :`".

- `textinput(`<`prompt1,`>` x1)` converts the input to a character string and assigns this string to the identifier `x1`. The default prompt is used, if no prompt string is specified.

- Several input values can be read with a single `textinput` command. Each identifier in the sequence of arguments makes `textinput` return a prompt, waiting for input to be assigned to the identifier. A character string preceeding the identifier in the argument sequence replaces the default prompt. Cf. example 3. Arguments that are neither prompt strings nor identifiers are ignored.

- Using a terminal version of MuPAD, the input must be terminated with the control character `<CTRL-D>` (the cursor has to be positioned behind the last character of the current input line). Graphical user interfaces of MuPAD may open a separate window for reading the input.

- The input may extend over several lines. In the output string, MuPAD uses the character `\n` (carriage return) to separate lines.

- Input characters with a leading \ are not interpreted as control characters, but as two separate characters.

- The identifiers `x1` etc. may have values. These are overwritten by `textinput`.

- `textinput` is a function of the system kernel.

Example 1. The default prompt is displayed, the input is converted to a character string and returned:

```
>> textinput()

 Please enter text input: << myinput >>

                              "myinput"
```

Example 2. A user-defined prompt is used, the input is assigned to the identifier `x`:

```
>> textinput("enter your name: ", x)

 enter your name: << Turing >>

                              "Turing"

>> x

                              "Turing"

>> delete x:
```

Example 3. If several values are to be read, separate prompts can be defined for each value:

```
>> textinput("She: ", hername, "He:  ", hisname)

 She: << Bonnie >>
 He:  << Clyde >>

                               "Clyde"

>> hername, hisname

                          "Bonnie", "Clyde"

>> delete hername, hisname:
```

`rtime, time` – measure real time and CPU time

`rtime()` returns the real time in milliseconds that elapsed since the start of the current MuPAD session.

`rtime(a1, a2, ...)` returns the real time needed to evaluate all arguments.

`time()` returns the total CPU time in milliseconds that was spent by the current MuPAD process.

`time(a1, a2, ...)` returns the CPU time needed by the current MuPAD process to evaluate all arguments.

Call(s):

- `rtime()`
- `rtime(a1, a2, ...)`
- `time()`
- `time(a1, a2, ...)`

Parameters:

`a1, a2, ...` — arbitrary MuPAD objects

Return Value: a nonnegative integer giving the elapsed time in milliseconds.

Related Functions: `prog::profile`

Details:

- The result of `rtime` is the real time. Thus, `rtime` can be used to measure the total time spent by the MuPAD process as well as by external processes spawned from inside the MuPAD session. Note, that an interactive call of `rtime()` is not very useful, since the idle time of the user is included. However, `rtime(a1, a2, ...)` often yields a useful and more realistic timing than `time(a1, a2, ...)` if the evaluation of the arguments spawns external processes. Such a situation may arise in a numerical computation because some routines of the `numeric` library call external numerical tools using hardware floats. Cf. example 4.

- The result of `time()` comprises all the computation time spent by the MuPAD process. This includes the time for system initialization and reading input (parsing). However, it excludes the time spent by other external processes, even if they were spawned from inside the MuPAD session or if they were started by a `system` command. Further, in an interactive session, the idle time between the execution of MuPAD commands is excluded.

- If no external process besides MuPAD are running, the timings returned by `rtime(a1, a2, ...)` and `time(a1, a2, ...)` roughly coincide.

- The time returned by `time` is computed in a system-dependent way, usually counting the number of clock ticks of the system clock. Hence, the result is a multiple of the system's time unit and cannot be more precise than 1 such unit. E.g., the time unit is 10 milliseconds for many UNIX systems.

- On computers without "time-sharing", such as the Macintosh, real time and CPU time roughly coincide.

- `rtime` and `time` are functions of the system kernel.

Example 1. This example shows how to do a time measurement and assign the computed value to an identifier at the same time. Note that the assignment needs extra parenthesis when passed as argument:

```
>> time((a := int(exp(x)*sin(x), x)))

                              220

>> a

                  sin(x) exp(x)   cos(x) exp(x)
                  ------------- - -------------
                        2               2

>> delete a:
```

Alternatively, one may time groups of statements in the following way:

```
>> t0 := time():
   command1
   command2
   ...
   time() - t0
```

Example 2. Here we use `rtime` to compute the elapsed hours, minutes and seconds since this session was started:

```
>> t := rtime()/1000:
   h := trunc(t/3600):
   m := trunc(t/60 - h*60):
   s := trunc(t - m*60 - h*3600):

>> print(Unquoted, "This session is running for " .
                   h . " hours, " .
                   m . " minutes and " .
                   s . " seconds.")

 This session is running for 0 hours, 0 minutes and 10 seconds.

>> delete t, h, m, s:
```

Example 3. To obtain a nicer output, the measured time can be multiplied with the appropriate time unit:

```
>> time((a := isprime(2^1000 - 1)))*msec
```

```
                         700 msec

>> time((a := isprime(2^1000 - 1)))*sec/1000.0

                          0.7 sec

>> delete a:
```

Example 4. The routine `numeric::inverse` for inverting numerical matrices tries to use an external hardware floating point tool. Assuming this tool to be available, the timings for inverting a large matrix may be as follows:

```
>> A := linalg::hilbert(300):
   time(numeric::inverse(A))*msec,
   rtime(numeric::inverse(A))*msec

                  1210 msec, 2013 msec
```

The real time MuPAD needs for sending the matrix and receiving the inverse is about 1.2 seconds. Ignoring other external processes, the external floating point tool needs about 0.8 seconds to invert the matrix. This adds up to the time indicated by `rtime`.

```
>> delete A:
```

Background:

- On a UNIX system, the time is measured using a system call to the function `'time'` on that system.

Changes:

- In previous releases, the last three digits of the time returned by `rtime` were always 0, i.e., the precision of the time measured by `rtime` was only one second. Now, `rtime` produces more precise results.

`traperror` – trap errors

`traperror(object)` traps errors produced by the evaluation of `object`.

`traperror(object, t)` does the same. Moreover, it stops the evaluation if it is not finished after a real time of `t` seconds.

Call(s):

- `traperror(object)`
- `traperror(object, t)`

Parameters:

`object`	—	any MuPAD object
`t`	—	the time limit: a nonnegative integer

Return Value: a nonnegative integer.

Related Functions: `error`, `prog::error`, `lasterror`

Details:

- `traperror` traps errors caused by the evaluation of the object. Syntactical errors, i.e., errors on parsing the object, cannot be caught. The same holds true for fatal errors causing the termination of MuPAD.
- `traperror` returns the error code `0` if no error happened. The error code is `1320` if the given time limit `t` is exceeded ('`Execution time exceeded`'). The error code is `1028` if the error was raised by the command `error`.
- If `traperror` has no time limit set and an '`Execution time exceeded`' error is raised by an enclosing `traperror(..., t)` command, then this error is not trapped by the inner `traperror`. It is trapped by the `traperror` call that has set the time limit. Cf. example 4.
- The object can be an assignment which, for syntactical reasons, must be enclosed in additional brackets. The following code fragment demonstrates a typical application of `traperror`:

```
if traperror((x := SomeErrorProneFunction())) = 0 then
     DoSomethingWith(x);
else RespondToTheError();
end_if;
```

- Use `lasterror` to reproduce the trapped error.
- `traperror` is a function of the system kernel.

Example 1. Errors that happen during the execution of kernel functions have various error codes, depending on the problem. E.g., '`Division by zero`' produces the error code 1025:

```
>> y := 1/x: traperror(subs(y, x = 0))

                                 1025
```

```
>> lasterror()

 Error: Division by zero [_power]
```

The following attempt to compute a huge floating point number fails because of numerical overflow. The corresponding error code is 20:

```
>> traperror(exp(12345678.9))

                                    20

>> lasterror()

 Error: Overflow/underflow in arithmetical operation;
 during evaluation of 'exp::float'
```

Example 2. All errors raised using the function `error` have the error code `1028`. Errors during the execution of library functions are of this kind:

```
>> traperror(error("My error!"))

                                   1028

>> lasterror()

 Error: My error!
```

Example 3. We try to factor a polynomial, but give up after ten seconds:

```
>> traperror(factor(x^1000 + 4*x + 1), 10)

                                   1320

>> lasterror()

 Error: Execution time exceeded;
 during evaluation of 'faclib::univ_mod_gcd'
```

Example 4. Here we have two nested `traperror` calls. The inner call contains an unterminated loop and the outer call has a time limit of `2` seconds. When the execution time is exceeded, this special error is not trapped by the inner `traperror` call. Because of the error, `print(1)` is never executed:

```
>> traperror((traperror((while TRUE do 1 end)); print(1)), 2)
```

```
>> lasterror()

 Error: Execution time exceeded
```

---

type – **the type of an object**

`type(object)` returns the type of the object.

Call(s):

- `type(object)`

Parameters:

`object` — any MuPAD object

Return Value: a domain type of type `DOM_DOMAIN` or a character string.

Overloadable by: `object`

Related Functions: `coerce`, `domtype`, `hastype`, `testtype`, `Type`

Details:

- If `object` is not an expression of domain type `DOM_EXPR`, then `type(object)` is equivalent to `domtype(object)`, i.e., `type` returns the domain type of the object.
- If `object` is an expression of domain type `DOM_EXPR`, then its type is determined by its 0-th operand (the "operator"). If the operator has a `"type"` slot, then `type` returns this value, which usually is a string. If the operator has no `"type"` slot, then `type` returns the string `"function"`.
- In contrast to most other functions, `type` does not flatten arguments that are expression sequences. Cf. example 4.
- `type` is a function of the system kernel.

Example 1. If an object is not an expression, its type equals its domain type:

```
>> type(3)

                              DOM_INT
```

Example 2. The operator of a sum is `_plus`; the type slot of that operator is `"_plus"`:

```
>> type(x + y*z)
```

"_plus"

`type` evaluates its argument: thereby, the difference of `x` and `y` becomes the sum of `x` and `(-1)*y`. Its type is not `"_subtract"`, but `"_plus"`:

```
>> type(x - y)
```

"_plus"

Example 3. If the operator of an expression is not a function environment having a type slot, the expression is of type `"function"`:

```
>> type(f(2))
```

"function"

Example 4. The following call to `type` is *not* regarded as a call with two arguments, because expression sequences in the argument are not flattened:

```
>> type((2, 3))
```

"_exprseq"

---

unassume – delete the properties of an identifier

`unassume(x)` deletes the properties of the identifier `x`.

Call(s):

- `unassume(x)`
- `unassume(`<*Global*>`)`

Parameters:

`x` — an identifier or a list or a set of identifiers

Options:

Global — deletes the "global property"

Return Value: the void object `null()`.

Related Functions: `assume`, `delete`, `getprop`, `is`

Details:

- `unassume` serves for deleting properties of identifiers set via `assume`. See `?property` for a short description of the property mechanism.
- If `x` is a list or a set of identifiers, then the properties of all specified identifiers are deleted.
- The calls `unassume()` and `unassume(`*Global*`)` are equivalent. This deletes the "global property" which is used for all identifiers. See `assume` for details on setting a global property.
- The command `delete x` deletes the value and the properties of the identifier `x`.

Example 1. Properties are attached to the identifiers `x` and `y`:

```
>> assume(x > 0): assume(y < 0): getprop(x), getprop(y)

                         > 0, < 0

>> sign(x), sign(y)

                          1, -1
```

`unassume` or `delete` deletes the properties:

```
>> unassume(x): delete y: getprop(x), getprop(y)

                          x, y

>> sign(x), sign(y)

                     sign(x), sign(y)
```

The properties of several identifiers can be deleted simultaneously by passing a list or a set to `unassume`:

```
>> assume(x > y): unassume([x, y]): getprop(x), getprop(y)

                          x, y
```

Example 2. All identifiers are assumed to represent real numbers. We set the corresponding global property:

```
>> assume(Global, Type::Real): getprop(x), getprop(y), getprop(z)

               Type::Real, Type::Real, Type::Real
```

```
>> Re(x), Im(y), Re(x*y*z)

                          x, 0, x y z
```

`unassume()` or `unassume(Global)` deletes the global property:

```
>> unassume(): Re(x), Im(y), Re(x*y*z)

                    Re(x), Im(y), Re(x y z)
```

`universe` – the set-theoretical universe

`universe` represents the set-theoretical universe of all objects.

Related Functions: `_union`, `_intersect`, `_minus`, `DOM_SET`

Details:

- `universe` is the only element of the domain `stdlib::Universe`.
- The standard set operations such as union, intersection and subtraction can be used with `universe`.

Example 1. We show some basic set operations involving `universe`:

```
>> universe union {a}

                           universe
```

```
>> universe intersect {a}

                              {a}
```

```
>> {a} minus universe

                              {}
```

`unloadmod` – unload a module

`unloadmod("modulename")` unloads the dynamic module named `modulename`.

`unloadmod()` tries to unload all currently loaded dynamic modules.

Call(s):

- `unloadmod("modulename"` $<$`,` *`Force`*$>$`)`
- `unloadmod()`

Parameters:

`"modulename"` — the name of a module: a character string

Options:

`Force` — forces the module manager to unload a *static* module.

Return Value: the void object of type `DOM_NULL`.

Side Effects: Unloading the machine code of a module does not affect the module domain. Accessing this module domain, the machine code of the corresponding module is reloaded automatically if needed. The function `reset` unloads all dynamic modules.

Further Documentation: Dynamic Modules - User's Manual and Programming Guide for MuPAD 1.4, Andreas Sorgatz, Oct 1998, Springer Verlag, Heidelberg, with CD-ROM, ISBN 3-540-65043-1.

Related Functions: `external`, `loadmod`, `module::displace`, `module::new`, `unexport`

Details:

- `unloadmod("modulename")` unloads the machine code of the module from the MuPAD process and the main memory.
- `unloadmod` produces an error if one tries to unload a *static* module without using the option *`Force`*.
- `unloadmod` is a function of the system kernel.

Example 1. Dynamic modules can be unloaded at runtime to save memory resources or to change and re-compile the modules (rapid prototyping).

```
>> loadmod("stdmod"): unloadmod():
```

After unloading, the machine code is reloaded automatically if needed:

```
>> stdmod::which("stdmod")

 "/usr/local/mupad/linux/modules/stdmod.mdm"
```

Background:

- The kernel functions `external`, `loadmod`, and `unloadmod` provide basic tools for accessing modules. Extended facilities are available with the `module` library.
- When calling a module function after its machine code was unloaded or displaced, the corresponding machine code is reloaded automatically. Here, in contrast to reloading the module using the function `loadmod`, the module domain is not affected.
- Some operating systems do not support unloading machine code at runtime. This, however, does not affect the usability of dynamic modules in any way.

`unprotect` – remove protection of identifiers

`unprotect(x)` removes any write protection of the identifier `x`.

Call(s):

- `unprotect(x)`

Parameters:

`x` — an identifier

Return Value: the previous protection level of `x`: either *ProtectLevelError* or *ProtectLevelWarning* or *ProtectLevelNone* (see `protect`).

Related Functions: `protect`

Details:

- `unprotect(x)` is equivalent to `protect(x, ProtectLevelNone)`.
- `unprotect` does not evaluate its argument. Cf. example 2.

Example 1. `unprotect` allows to assign values to system functions:

```
>> unprotect(sign): sign(x) := 1
```

$$1$$

However, we strongly advise not to change identifiers protected by the system. We undo the previous assignment:

```
>> delete sign(x): protect(sign, ProtectLevelError):
```

Example 2. `unprotect` does not evaluate its argument. Here the identifier `x` is unprotected and not its value `y`:

```
>> x := y:  protect(y): unprotect(x): y := 1

 Warning: protected variable y overwritten
```

$$1$$

```
>> unprotect(y): delete x, y:
```

Changes:

- The options *None*, *Warning* and *Error* of `protect` and thus the return values of `unprotect` were renamed to *ProtectLevelNone*, *ProtectLevelWarning* and *ProtectLevelError*.

`userinfo` – print progress information

`userinfo(n, message)` prints a message if an information level larger or equal to `n` is set via `setuserinfo`.

`userinfo(n1..n2, message)` prints a message if the information level set by `setuserinfo` is between `n1` and `n2`.

Call(s):

- userinfo(<*Text*,> <*NoNL*,> n, message1, message2, ...)
- userinfo(<*Text*,> <*NoNL*,> n1..n2, message1, message2, ...)

Parameters:

n, n1, n2	—	the information levels: positive integers
message1, message2, ...	—	arbitrary MuPAD objects. Typically, character strings.

Options:

Text	—	do not separate the arguments by commas in the output
NoNL	—	do not separate the arguments by commas in the output, do not start a new line after the output, and do not precede the output by the string "Info: ".

Return Value: the void object of type DOM_NULL.

Side Effects: The formatting of the output of userinfo is sensitive to the environment variable TEXTWIDTH.

Related Functions: print, setuserinfo, warning

Details:

- userinfo must not be used on the interactive level. It should be built into the body of a procedure or of a domain method to print status information such as the chosen algorithm, intermediate results etc. If a userinfo command is built into a procedure by the name f, say, then it is activated by setting an appropriate information level via setuserinfo(f, n). The information is printed during subsequent calls to f.
- The print output consists of the evaluation of the message arguments, possibly followed by the name of the procedure (see the function setuserinfo). Strings are printed without quotes. The pretty printer is not used. Unless one of the options *Text* or *NoNL* is given, the message arguments are separated by commas in the output. Unless the option *NoNL* is given, the print output is preceded by the string "Info: " and a new line is started after the output.
- The information level of a single procedure, of all procedures of a domain, or, of all procedures in general can be specified using setuserinfo. All three levels may apply to a procedure simultaneously.
- Most of the functions in the MuPAD library provide status information via userinfo. See example 3.
- userinfo is a function of the system kernel.

Example 1. The function `expr2text` is useful for incorporating MuPAD objects in a text message:

```
>> f := proc(x)
        begin
          userinfo(2, "the argument is " . expr2text(x));
          x^2
        end_proc:

>> setuserinfo(f, 2, Name): f(12)

 Info: the argument is 12 [f]

                               144

>> setuserinfo(f, 0): delete f:
```

Example 2. A call of the form `userinfo(n, message)` causes `message` to be displayed if the information level is at least as high as `n`. If you want `message` to be displayed only if the information level equals `n`, use a range that consists of one point only:

```
>> f := proc()
        begin
          userinfo(2..2, "Infolevel = 2");
          userinfo(2, "Infolevel >= 2");
        end_proc:

>> setuserinfo(f, 2): f():

 Info: Infolevel = 2
 Info: Infolevel >= 2

>> setuserinfo(f, 3): f():

 Info: Infolevel >= 2

>> setuserinfo(f, 0): delete f:
```

Example 3. By setting the information level of `faclib` to 5, we get information on the algorithms used for factorization:

```
>> setuserinfo(faclib, 5): factor(x^2 + 2*x + 1)
```

```
 Info: faclib::monomial called with poly(x^2 + 2*x + 1, [x])
 Info: Squarefree factorization (Yun's algorithm) called

                                        2
                                 (x + 1)

>> setuserinfo(faclib, 0):
```

Background:

- `userinfo` does not evaluate the messages unless they are printed.
- The global table of information levels for all procedures can be obtained by the call `setuserinfo()`.

Changes:

- The option *NoNL* was added.

`val` – the value of an object

`val(object)` replaces every identifier in `object` by its value.

Call(s):

- `val(object)`

Parameters:

`object` — any MuPAD object

Return Value: the "evaluated" object.

Related Functions: `eval`, `hold`, `level`, `LEVEL`, `MAXLEVEL`

Details:

- `val` does not perform any simplification of the result.
- If the result of `val` is a set, duplicate elements are removed from that set.
- `val` does not work recursively, i.e., if the value of an identifier in turn contains identifiers, then these are not replaced by their values. See example 3.

- val does not flatten its argument. Hence, an expression sequence is accepted as argument. Cf. example 2.
- val is a function of the system kernel.

Example 1. val replaces identifiers by their values, but does not call arithmetical functions such as _plus:

```
>> a := 0: val(a*b + 4 + 0)

                        0 b + 4 + 0
```

Duplicate elements in sets are removed:

```
>> a := b: val({a, b, a*0})

                        {b, 0 b}

>> delete a:
```

Example 2. val does not flatten its argument, nor does it remove void objects of type DOM_NULL:

```
>> a := null(): val((a, null()))

                        null(), null()
```

However, it is not legal to pass several arguments:

```
>> val(a, null())

 Error: Wrong number of arguments [val]

>> delete a:
```

Example 3. val does not recursively substitute values for the identifiers:

```
>> delete a, b: a := b: b := c: val(a)

                        b
```

---

version – the version number of the MuPAD library

version() returns the version number of the installed MuPAD library.

Call(s):

- `version()`

Return Value: the version number: a list of three nonnegative integers.

Related Functions: `patchlevel`, `Pref::kernel`

Details:

- The call `Pref::kernel()` returns the version number of the installed MuPAD kernel.
- The version numbers of the kernel and the library may differ: `version` refers to the library, whereas the call `Pref::kernel()` returns the version number of the kernel.

Example 1. The version of this MuPAD library is:

```
>> version()

                      [2, 5, 0]
```

`warning` – print a warning message

`warning(message)` prints the warning `message`.

Call(s):

- `warning(message)`

Parameters:

`message` — a character string

Return Value: the void object of type `DOM_NULL`.

Side Effects: The formatting of the output of `warning` is sensitive to the environment variable `TEXTWIDTH`.

Related Functions: `error`, `print`, `userinfo`

Details:

- `warning(message)` prints the message with the prefix "`Warning: `".
- `warning` may be used to print information about potential problems in an algorithm. E.g., it is used in `limit` to provide hints. Cf. example 3.
- `warning` is a function of the system kernel.

Example 1. A warning:

```
>> warning("You should not do this!"):

 Warning: You should not do this!
```

Example 2. This example shows a simple procedure which divides two numbers. If the second argument is omitted, a warning is printed and the computation continues:

```
>> mydivide := proc(x, y)
   begin
     if args(0) < 2 then
       warning("Denominator not given, using 1.");
       y := 1;
     end_if:
     x/y
   end_proc:
   mydivide(10)

 Warning: Denominator not given, using 1. [mydivide]

                                10
```

Example 3. The following integral cannot be written in closed form for general values of `a` and `b`:

```
>> int(1/x, x=a..b)

 Warning: Found potential discontinuities of the antiderivative.
 Try option 'Continuous' or use properties (?assume). [intlib::\
 antiderivative]

                           / 1            \
                        int| -, x = a..b |
                           \ x            /
```

The user can react to the warning by assuming some properties for `a` and `b`:

```
>> assume(1<a<b):
   int(1/x, x=a..b)

                              ln(b) - ln(a)

>> assume(a<0<b): int(1/x, x=a..b)

                                undefined

>> unassume({a, b}):
```

`write` – write the values of variables into a file

`write(filename)` stores all assigned identifiers of the MuPAD session with their current values in a file specified by `filename`.

`write(filename, x1, x2, ...)` stores the current values of the identifiers `x1`, `x2` etc.

`write(n)` and `write(n, x1, x2, ...)` store the data in the file associated with the file descriptor `n`.

Call(s):

- `write(<format>, filename)`
- `write(<format>, filename, x1, x2, ...)`
- `write(n)`
- `write(n, x1, x2, ...)`

Parameters:

`filename`	—	the name of a file: a character string
`x1, x2, ...`	—	identifiers
`n`	—	a file descriptor provided by `fopen`: a nonnegative integer

Options:

`format`	—	the write format: either *Bin* or *Text*. With *Bin*, the data are stored in MuPAD's binary format. With *Text*, standard ASCII format is used. The default is *Bin*.

Return Value: the void object of type `DOM_NULL`.

Side Effects: The function is sensitive to the environment variable `WRITEPATH`. If this variable has a value, the file is created in the corresponding directory. Otherwise, the file is created in the "working directory".

Related Functions: `fclose`, `fileIO`, `finput`, `fname`, `fopen`, `fprint`, `fread`, `ftextinput`, `pathname`, `print`, `protocol`, `read`, `READPATH`, `WRITEPATH`

Details:

- `write` serves for storing information from the current MuPAD session in a file. The file contains the values of identifiers of the current session. These identifiers are assigned the stored values when this file is read into another MuPAD session via the function `read`.

- The file may be specified directly by its name. In this case, `write` creates a new file or overwrites an existing file; `write` opens and closes the file automatically.

  If `WRITEPATH` does not have a value, `write` interprets the file name as a pathname relative to the "working directory".

  Note that the meaning of "working directory" depends on the operating system. On Windows systems, the "working directory" is the folder where MuPAD is installed. On UNIX or Linux systems, it is the current working directory in which MuPAD was started.

  On the Macintosh, an empty file name may be given. In this case, a dialogue box is opened in which the user can choose a file. Further, on the interactive level, MacMuPAD warns the user, if an existing file is about to be overwritten.

  Also absolute path names are processed by `write`.

- Instead of a file name, also a file descriptor of a file opened via `fopen` can be used. Cf. example 2. In this case, the data written by `write` are appended to the corresponding file. The file is not closed automatically by `write` and must be closed by a subsequent call to `fclose`.

  Note that `fopen(filename)` opens the file in read-only mode. A subsequent `write` command to this file causes an error. Use the ***`Write`*** or ***`Append`*** option of `fopen` to open the file for writing.

  The file descriptor `0` represents the screen.

- `write` stores the *values* of the given identifiers, not *their full evaluation*! Cf. example 3.

  NOTE

- For an overview of all file related MuPAD functions, also try `?fileIO`.

- `write` is a function of the system kernel.

Option <Text>:

- In ASCII format, assignments of the form

  `sysassign(identifier, hold(value)):`

  are written into the file. Cf. example 1.

Example 1. The variable `a` and its value `b + 1` are stored in a file named `test`:

```
>> a := b + 1: write(Text, "test", a):
```

The content of this file is displayed via `ftextinput`:

```
>> ftextinput("test")

                    "sysassign(a, hold(b + 1)):"
```

We delete the value of `a`. Reading the file `test` restores the previous value:

```
>> delete a: read("test"): a

                                  b + 1

>> delete a:
```

Example 2. The file `test` is opened for writing using MuPAD's binary format:

```
>> n := fopen("test", Write)

                                   17
```

This number is the descriptor of the file and can be used in a write command:

```
>> a := b + 1: write(n, a):

>> delete a: read("test"): a

                                  b + 1
```

We close the file and clean up:

```
>> fclose(n): delete n, a:
```

Example 3. The value `b + 1` is assigned to the identifier `a`. After assigning the value `2` to `b`, complete evaluation of `a` yields `3`:

```
>> a := b + 1: b := 2: a

                                    3
```

Note, however, that the value of `a` is the expression `b + 1`. This value is stored by a `write` command:

```
>> write(Text, "test", a): ftextinput("test")

                    "sysassign(a, hold(b + 1)):"
```

Consequently, this value is restored after reading the file into a MuPAD session:

```
>> delete a, b: read("test"): a

                                  b + 1

>> delete a:
```

`zeta` – the Riemann zeta function

`zeta(z)` represents the Riemann zeta function $\zeta(z) = \sum_{k=1}^{\infty} k^{-z}$.

Call(s):

- `zeta(z)`

Parameters:

`z` — an arithmetical expression

Return Value: an arithmetical expression.

Overloadable by: `z`

Side Effects: When called with a floating point argument, the function is sensitive to the environment variable `DIGITS` which determines the numerical working precision.

Related Functions: `bernoulli`

Details:

- The zeta function is defined for all complex arguments except for the simple pole $z = 1$.
- A floating point result is returned for floating point arguments.
- The following special values are implemented: $\zeta(z) = 0$ for even integers $z < 0$, $\zeta(z) = -$`bernoulli`$(z)/(1-z)$ for odd integers $z < 0$, $\zeta(0) = -1/2$, $\zeta(z) = (2\pi)^z|$`bernoulli`$(z)|/2/z!$ for even integers $z > 0$.

  `zeta` returns an unevaluated function call, if the argument does not evaluate to one of the above numbers.
- The float attribute of `zeta` is a kernel function, i.e., floating point evaluation is fast. Use `zeta(float(z))` rather than `float(zeta(z))`, because the evaluation of the intermediate exact result `zeta(z)` may be costly for integers `z` of large absolute value.

NOTE

Example 1. We demonstrate some calls with exact and symbolic input data:

```
>> zeta(-6), zeta(-5), zeta(-4), zeta(-3), zeta(-2), zeta(-1)

                 0, -1/252, 0, 1/120, 0, -1/12

>> zeta(0), zeta(2), zeta(3), zeta(4), zeta(5), zeta(6), zeta(7)

               2              4              6
             PI             PI             PI
       -1/2, ---, zeta(3), ---, zeta(5), ---, zeta(7)
              6              90             945

>> zeta(1/2), zeta(1 + I), zeta(z^2 -I)

                                                2
              zeta(1/2), zeta(1 + I), zeta(z  - I)
```

Floating point values are computed for floating point arguments:

```
>> zeta(-1001.0), zeta(12.3), zeta(0.5 + 14.13472514*I)

 -1.348590824e1771, 1.000199699,

    0.0000000002163160213 - 0.000000001358779595 I
```

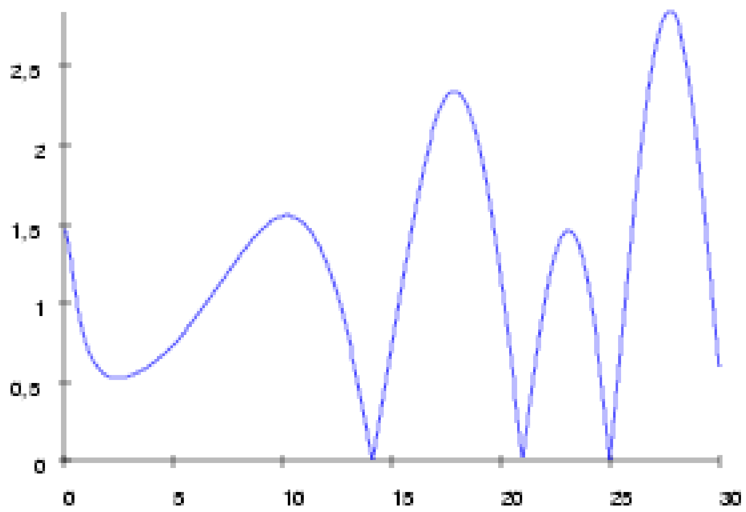
`zeta` has a pole at the point $z = 1$:

```
>> zeta(1)

 Error: singularity [zeta]
```

Example 2. Looking for roots of the Zeta function, we plot the function $f(z) = |\zeta(z)|$ along the "critical line" of complex numbers with real part $1/2$:

```
>> plotfunc2d(Labels = ["", ""], Title = "", Grid = 500,
              abs(zeta(1/2 + y*I)), y = 0..30)
```



The following procedure is a simple implementation of the usual Newton method for finding numerical roots of ζ. Note that numeric differentiation is used within the Newton step, because MuPAD does not provide a symbolic derivative of zeta:

```
>> NewtonStep := proc(z)
   local h, f, f2, fprime;
   begin
     z := float(z);
     h := 10^(-DIGITS/2.0)*(1 + abs(z));
     f := zeta(z);
     f2 := zeta(z + h);
     fprime := (f2 - f)/h;
     return(z - f/fprime)
   end_proc:
```

The sequence `z:=NewtonStep(z)` converges to a root, if the initial value is a sufficiently good approximation of the root:

```
>> z:= 1/2 + 21*I:  z := NewtonStep(z): z, abs(zeta(z))
```

```
          0.5002926366 + 21.0220145 I, 0.0003338475592

>> z := NewtonStep(z): z, abs(zeta(z))

        0.4999999108 + 21.02203966 I, 0.0000001039451698

>> z := NewtonStep(z): z, abs(zeta(z))

                0.5 + 21.02203964 I, 1.387291733e-11

>> delete NewtonStep, z:
```

`zip` – combine lists

`zip(list1, list2, f)` combines two lists via a function `f`. It returns a list whose i-th entry is `f(list1[i], list2[i])`. Its length is the minimum of the lengths of the two input lists.

`zip(list1, list2, f, default)` returns a list whose length is the maximum of the lengths of the two input lists. The shorter list is padded with the `default` value.

Call(s):

- `zip(list1, list2, f)`
- `zip(list1, list2, f, default)`

Parameters:

`list1, list2`	—	lists of arbitrary MuPAD objects
`f`	—	any MuPAD object. Typically, a function of two arguments.
`default`	—	any MuPAD object

Return Value: a list.

Overloadable by: `list1, list2`

Related Functions: `map`, `op`, `select`, `split`

Details:

- If `f` produces the void object of type `DOM_NULL`, then this element is removed from the resulting list.
- `zip` is recommended for fast manipulation of lists. It is a function of the system kernel.

Example 1. The fastest way of adding the entries of two lists is to 'zip' them via the function `_plus`:

```
>> zip([a, b, c, d], [1, 2, 3, 4], _plus)
```

```
                    [a + 1, b + 2, c + 3, d + 4]
```

If the input lists have different lengths, then the shorter list determines the length of the returned list:

```
>> zip([a, b, c, d], [1, 2], _plus)
```

```
                           [a + 1, b + 2]
```

The longer list determines the length of the returned list if a value for padding the shorter list is provided:

```
>> zip([a, b, c, d], [1, 2], _plus, 17)
```

```
                   [a + 1, b + 2, c + 17, d + 17]
```