From MuPAD 1.4 to MuPAD 2.0

written by

Klaus Drescher, SciFace Software GmbH, drescher@sciface.com

and

Winfried Fakler, University of Paderborn, fakler@mupad.de

This document summarizes the most important changes from MuPAD 1.4 to MuPAD 2.0 including the changes in the library and the programming language. It also describes various tools to support users in converting code written for version 1.4 to version 2.0.

Contents

1 Introduction

Version 2.0 of MuPAD differs significantly from the previous versions. The new version features enhanced user interfaces, a new debugger frontend for the MuPAD Pro version under Windows, the support of dynamic modules, a more detailed documentation, and many other things.

In this document, the changes concerning the mathematical power of MuPAD are described. This power resides in the library as well as in the programming language provided by the system. Both have changed significantly.

Some *library functions* were renamed or moved from one library package to another in order to achieve a more consistent naming and a more systematic library structure. Other functions still exist, but have a changed or enhanced functionality. Many new functions were introduced with version 2.0. An overview of all library changes is given in the chapters

- What is new in the MuPAD 2.0 Library?
- What has changed in the MuPAD 2.0 Library?

With the introduction of lexical scoping in version 2.0, the *programming language* has changed. The domains concept has got its own language constructs and is now completely integrated into the MuPAD language. A detailed description together with the new grammar is given in the chapter

- The new MuPAD Language

of this document. Reading this particular chapter is essential for anybody writing MuPAD code.

Due to the changes in the library and the programming language, it may happen that existing MuPAD code written for previous versions needs to be adapted to the new syntax. For this conversion, various *support* facilities are available. They are described in the chapter

- Version Changing Support

Some of the most noticeable changes in the *library* are the completely redesigned property mechanism and the introduction of conditionally defined objects which are integrated into the solver concept. This significantly influences MuPAD's solve facilities as well as various system functions such as the integrator `int` or `limit`. As an example, let us solve the general linear equation $ax + b = 0$ with respect to x. In version 2.0, `solve` produces a complete answer considering all possible cases:

```
>> solve(a*x + b = 0, x)
```

```
          / {   b }
 piecewise| { - - } if a <> 0, C_ if a = 0 and b = 0,
          \ {   a }

                              \
    {} if a = 0 and b <> 0    |
                              /
```

The result of an integration may depend on assumptions on the parameters and the integration variable. E.g., for the indefinite integral $\int 1/\sqrt{ax^2+bx+c}\,\mathrm{d}x$:

```
>> assume(a > 0): int(1/sqrt(a*x^2 + b*x + c), x)
```

```
                                              2  1/2
            ln(b + 2 a x + 2 (a (c + b x + a x ))   )
            ----------------------------------------
                              1/2
                             a
```

```
>> assume(a < 0): int(1/sqrt(a*x^2 + b*x + c), x)
```

```
                            /    b + 2 a x    \
                      arcsin| --------------- |
                            |   2         1/2 |
                            \ (b  - 4 a c)    /
                     - ------------------------
                                  1/2
                             (-a)
```

2 Version Changing Support

2.1 Utilities in the New Library

There is a "compatibility mode" between the MuPAD version 1.4 and 2.0. With the call `prog::changes()`, "dummy" versions of functions existing in 1.4 and not existing in 2.0 are created and added to the 2.0 library. Whenever such "old" functions are called, they notify you about the fact that they are obsolete. If a corresponding new function with the same functionality exists, it is called automatically. Thus, you stand a fair chance that your old code runs in version 2.0, even if it uses library functions that do not exist any more. However, the warnings give you a hint, which calls inside your old code should be adapted to the new library. See the help page of `prog::changes` for further details.

A special preference `Pref::warnChanges` was introduced with version 2.0 to support the porting of old MuPAD code to the new version. After calling `Pref::warnChanges(TRUE)`, old code running in version 2.0 produces warnings if environment variables such as `DIGITS` are used in a way that is not consistent with the new kernel. The preference `warnChanges` is set to `FALSE` by default. It may enabled by a call `Pref::warnChanges(TRUE)` during a session. Alternatively, on UNIX or LINUX systems, the command line option `-P W` may be used when calling MuPAD as in `xmupad -P W`.

2.2 Converting Programs

A command line program is provided which converts MuPAD programs written for version 1.4.x to version 2.0. It parses files containing old MuPAD code and makes syntactical changes nesseccary to run the programs under version 2.0. In particular, it converts pure functions defined by `fun` and `func` into procedures (`fun` and `func` are obsolete and do not exist in 2.0 anymore). Additionally, it automatically converts certain obsolete names. For example, the identifier `this` is replaced by the new special variable `dom`. The functions `funcattr` and `domattr` are changed to `slot`. Also names of renamed library functions are adapted.

The name of the command line program is `version20` (`version20.exe` under Windows). You find it in the directory where the MuPAD binaries are installed. At the same location, you find a brief help file for the program in HTML format, called `version20.html`.

The program must be run from the command line. Its syntax is

`version20` *`[-h] [-v] [-s] [-l`* *logfile] [`-o`* *outfile] [infile]*

The options are described in `version20.html`.

Please note that the semantics of some MuPAD procedures has changed with version 2.0. *The conversion program* `version20` *cannot check whether the semantics of the translated programs is still correct!* It is only an aid for the *syntactical* changes. *It is necessary to inspect the translated programs manually to detect any semantical problems!*

It is strongly recommended to create a log file using the option `-l`. It will contain information about all changes which `version20` applied to your code. The log file should be inspected carefully.

3 What is new in the MuPAD 2.0 Library?

The MuPAD 2.0 library has many new features. For example, the plot library was completely rewritten and enhanced with new features.

In the following, the new functions are listed in tables grouped according to various mathematical areas:

- Basic Mathematics and Calculus
- Special Functions
- Graphics
- Linear Algebra
- Linear Optimization
- Numerics
- Statistics and Probability
- Generating Fortran Code
- Utilities
- Data Types, Domains and Categories

3.1 Basic Mathematics and Calculus

New Function	Description
`detools::arbFuns`	number of arbitrary functions in the general solution of an involutive partial differential equation
`detools::autoreduce`	autoreduction of a system of differential equations
`detools::cartan`	Cartan characters of a differential equation

New Function	Description
`detools::characteristics`	characteristics of partial differential equation
`detools::charODESystem`	characteristic system of partial differential equation
`detools::charSolve`	solve partial differential equation with the method of characteristics
`detools::detSys`	determining system for Lie point symmetries
`detools::derList2Tree`	minimal tree with a given list of derivatives as leaves
`detools::euler`	Euler operator of variational calculus
`detools::hasHamiltonian`	check for Hamiltonian vector field
`detools::hasPotential`	check for gradient vector field
`detools::hilbert`	Hilbert polynomial of a differential equation
`detools::modode`	modified equation
`detools::ncDetSys`	determining system for non-classical Lie symmetries
`detools::pdesolve`	solver for partial differential equations
`detools::transform`	change of variables for differential equations
`ground`	ground term (zeroth coefficient) of a polynomial
`intlib::byparts`	perform integration by parts
`matrix`	definition of matrices and vectors
`polylib::cyclotomic`	cyclotomic polynomials
`polylib::divisors`	divisors of a polynomial, polynomial expression, or Factored element
`polylib::elemSym`	elementary symmetric polynomials
`polylib::makerat`	convert expression into rational function over a suitable field
`polylib::realroots`	isolate all real roots of a real univariate polynomial
`polylib::representByElemSym`	represent symmetric by elementary symmetric polynomials
`polylib::sortMonomials`	sorting monomials with respect to a term ordering
`polylib::splitfield`	the splitting field of a polynomial
`property::hasprop`	test whether an object has properties
`property::implies`	test whether one property implies another
`property::simpex`	simplify Boolean expressions
`solvelib::conditionalSort`	possible sortings of a list depending on parameters
`solvelib::getElement`	get one element of a set

New Function	Description
`solvelib::isFinite`	test whether a set is finite
`solvelib::preImage`	preimage of a set under a mapping
`solvelib::Union`	union of a system of sets
`student::plotRiemann`	plot of a numerical approximation to an integral using rectangles
`student::plotSimpson`	plot of a numerical approximation to an integral using Simpson's rule
`student::plotTrapezoid`	plot of a numerical approximation to an integral using the Trapezoidal rule
`student::riemann`	numerical approximation to an integral using rectangles
`student::simpson`	numerical approximation to an integral using Simpson's rule
`student::trapezoid`	numerical approximation to an integral using the Trapezoidal rule
`universe`	the set-theoretic universe

3.2 Special Functions

New Function	Description
`Ci`	the cosine integral function
`Ei`	the exponential integral function (used to be `eint`)
`arg`	the argument (polar angle) of a complex number
`besselI`, `besselJ`, `besselK`, `besselY`	the Bessel functions
`combinat::modStirling`	modified Stirling numbers
`lambertV`, `lambertW`	lower and upper real branch of the Lambert function
`log`	the logarithm to an arbitrary base
`numlib::invphi`	the inverse of the Euler phi function
`polylog`	the polylogarithm function
`signIm`	the sign of the imaginary part of a complex number

3.3 Graphics

New Function	Description
`plot`	display graphical objects on the screen
`plot::Curve2d`	graphical object for two-dimensional curve plots
`plot::Curve3d`	graphical object for three-dimensional curve plots
`plot::Ellipse2d`	graphical object for two-dimensional ellipses
`plot::Function2d`	graphical object for two-dimensional function plots
`plot::Function3d`	graphical object for three-dimensional function plots
`plot::Group`	graphical object representing a group of graphical objects
`plot::Point`	graphical object for two- and three-dimensional points
`plot::Pointlist`	graphical object for a list of either two- or three-dimensional points
`plot::Polygon`	graphical object for two- and three-dimensional polygons
`plot::Rectangle2d`	graphical object for two-dimensional rectangles
`plot::Scene`	graphical object for graphical scenes
`plot::Surface3d`	graphical object for three-dimensional surface plots
`plot::copy`	create a copy of a graphical primitive
`plot::HOrbital`	visualization of electron orbitals of a hydrogen atom
`plot::implicit`	implicit plot of smooth functions
`plot::inequality`	generate a 2D plot of inequalities
`plot::line`	graphical object for lines
`plot::modify`	create modified copies of graphical objects
`plot::ode`	plot the numerical solution of an ordinary differential equation
`plot::vector`	graphical object for vectors

3.4 Linear Algebra

New Function	Description
`linalg::companion`	the companion matrix of a univariate polynomial
`linalg::factorLU`	LU-decomposition of a matrix
`linalg::frobeniusForm`	Frobenius form of a matrix
`linalg::hessenberg`	Hessenberg matrix
`linalg::hilbert`	Hilbert matrix
`linalg::inverseLU`	computing the inverse of a matrix using LU-decomposition

New Function	Description
`linalg::invhilbert`	inverse of a Hilbert matrix
`linalg::minpoly`	minimal polynomial of a matrix
`linalg::permanent`	permanent of a matrix
`linalg::pseudoInverse`	Moore-Penrose inverse of a matrix
`linalg::smithForm`	Smith canonical form of a matrix
`linalg::substitute`	replace a part of a matrix by another matrix
`linalg::vandermondeSolve`	solve a linear Vandermonde system
`linalg::wiedemann`	solving linear systems by Wiedemann's algorithm
`matrix`	definition of matrices and vectors

3.5 Linear Optimization

New Function	Description
`linopt::corners`	return the feasible corners of a linear program
`linopt::plot_data`	plot the feasible region of a linear program
`linopt::Transparent`	return the ordinary simplex tableau of a linear program
`linopt::Transparent::autostep`	perform the next simplex step
`linopt::Transparent::clean_basis`	delete all slack variables of the first phase from the basis
`linopt::Transparent::convert`	transform the given tableau into a structure viewable on the screen
`linopt::Transparent::dual_prices`	get the dual solution belonging to the given tableau
`linopt::Transparent::phaseI_tableau`	start an ordinary phase one of a 2-phase simplex algorithm
`linopt::Transparent::phaseII_tableau`	start the phase two of a 2-phase simplex algorithm
`linopt::Transparent::result`	get the basic feasible solution belonging to the given simplex tableau
`linopt::Transparent::simplex`	runs the current phase of the 2-phase simplex algorithm to the end
`linopt::Transparent::suggest`	suggest the next step in the simplex algorithm

New Function	Description
`linopt::Transparent::userstep`	perform a user defined simplex step

3.6 Numerics

New Function	Description
`numeric::complexRound`	round a complex number towards the real or imaginary axis
`numeric::cubicSpline`	interpolation by cubic splines (used to be `spline`)
`numeric::expMatrix`	the exponential of a matrix
`numeric::fft`	Fast Fourier Transform (used to be `fft`)
`numeric::fMatrix`	functional calculus for numerical square matrices
`numeric::fsolve`	search for a numerical root of a system of equations
`numeric::indets`	search for indeterminates
`numeric::int`	numerical integration
`numeric::invfft`	inverse Fast Fourier Transform (used to be `ifft`)
`numeric::lagrange`	polynomial interpolation (used to be `lagrange`)
`numeric::linsolve`	solve a system of linear equations
`numeric::matlinsolve`	solve a linear matrix equation
`numeric::odesolve2`	numerical solution of an ordinary differential equation
`numeric::polyroots`	numerical roots of a univariate polynomial
`numeric::polysysroots`	numerical roots of a system of polynomial equations
`numeric::rationalize`	approximate a floating point number by a rational number (used to be `sharelib::rational`)
`numeric::realroot`	numerical search for a real root of a real univariate function
`numeric::realroots`	isolate *all* real roots a real univariate function
`numeric::sort`	sort a numerical list
`numeric::solve`	numerical solution of equations (the float slot of `solve`)
`numeric::spectralradius`	the spectral radius of a matrix (used to be `numeric::vonMises`)
`numeric::sum`	compute an infinite sum (the float slot of `sum`)

3.7 Statistics and Probability

New Function	Description
`stats::BPCorr`	Bravais-Pearson correlation
`stats::FCorr`	Fechner correlation
`stats::a_quantil`	alpha-quantile of discrete data
`stats::calc`	apply functions to samples
`stats::col`	select and re-arrange columns of a sample
`stats::concatCol`	concatenate samples column-wise
`stats::concatRow`	concatenate samples row-wise
`stats::geometric`	the geometric mean
`stats::harmonic`	the harmonic mean
`stats::kurtosis`	kurtosis (excess)
`stats::modal`	the modal (most frequent) value(s)
`stats::obliquity`	obliquity (skewness)
`stats::quadratic`	the quadratic mean
`stats::reg`	regression (general least square fit)
`stats::row`	select and re-arrange rows of a sample
`stats::sample`	the domain of statistical samples
`stats::sample2list`	convert a sample to a list of lists
`stats::selectRow`	select rows of a sample
`stats::sortSample`	sort the rows of a sample
`stats::tabulate`	statistics of duplicate rows
`stats::unzipCol`	extract columns from a list of lists
`stats::zipCol`	convert a sequence of columns to a list of lists

3.8 Generating Fortran Code

New Function	Description
`generate::Macrofort::closeOutputFile`	close FORTRAN file
`generate::Macrofort::genFor`	FORTRAN code generator
`generate::Macrofort::init`	initialize genFor
`generate::Macrofort::openOutputFile`	open FORTRAN file
`generate::Macrofort::setAutoComment`	automatic comments
`generate::Macrofort::setIOSettings`	set I/O settings
`generate::Macrofort::setOptimizedOption`	set optimization
`generate::Macrofort::setPrecisionOption`	set precision

3.9 Utilities

New Function	Description
NOTEBOOKFILE	Notebook file name
NOTEBOOKPATH	Notebook path
Pref::alias	controls the output of aliased expressions
Pref::ignoreNoDebug	controls debugging of procedures
Pref::timesDot	determine the output of products
Pref::typeCheck	type checking of formal parameters
Pref::warnChanges	warnings about changes wrt. the previous version of MuPAD
Pref::warnDeadProcEnv	warnings about wrong usage of lexical scope
Pref::warnLexProcEnv	warnings about usage of variables from lexical scope
Type::Arithmetical	a type representing arithmetical objects
Type::Property	type to identify properties
Type::Residue	a property representing a residue class
coerce	conversion of objects
delete	delete the value of an identifier
end	the keyword end
fp::unapply	create a procedure from an expression
in	membership
int2text	convert an integer to a character string
lasterror	reproduce the last error
lhs	the left hand side of equations, inequalities, relations and ranges
match	pattern matching
matchlib::analyze	structure of an expression
operator	define a new operator symbol
output::ordinal	ordinal numbers
output::tree	display of trees
package	load a package of new library functions
prog::allFunctions	overview of all functions
prog::changes	generate obsolete functions of MuPAD version 1.4
prog::check	checking MuPAD objects
prog::error	error message and internal error number
prog::find	find operands of expressions
prog::getname	the name of an object
prog::init	loading objects
prog::isGlobal	information about reserved identifiers
prog::traced	find traced functions

New Function	Description
`rhs`	the right hand side of equations, inequalities, relations and ranges
`share`	create a unique data representation
`slot`	method or entry of a domain or a function environment
`unexport`	undoes the export of library functions
`warning`	print a warning message
`%if`	conditional creation of code by the parser

3.10 Data Types, Domains and Categories

New Function	Description
`Cat::Set`	the category of sets of complex numbers
`Dom::Ideal`	the domains of sets of ideals
`Dom::ImageSet`	the domain of images of sets under mappings
`Dom::MonomOrdering`	monomial orderings
`Dom::MultivariatePolynomial`	the domains of multivariate polynomials
`Dom::SparseMatrixF2`	the domain of sparse matrices over the field with two elements
`Dom::UnivariatePolynomial`	the domains of univariate polynomials
`Factored`	domain of objects kept in factored form
`adt::Queue`	abstract data type “Queue”
`adt::Stack`	abstract data type “Stack”
`adt::Tree`	abstract data type “Tree”
`piecewise`	the domain of conditionally defined objects
`solvelib::BasicSet`	the basic infinite sets

4 What has changed in the MuPAD 2.0 Library?

This chapter lists the library changes from version MuPAD 1.4 to MuPAD 2.0.

The table consists of two columns. In the first column, you find the names of functions or environment variables of version 1.4.

Some of the functions listed here have been renamed in MuPAD 2.0 for consistency. For these functions, the second column lists the new name of the corresponding function.

Other functions have been removed completely or subsumed into some other function. In these cases, the second column directs you at some function you may use instead.

For the other functions listed here, the functionality has been extended substantially or the calling syntax has changed (again, for consistency). Please refer to the corresponding documentation for further details.

Name in 1.4	New Name & Changed Features
`:=`, `_assign`	changed
`$`, `_seqgen`	changed
`Cat::CommutativeRing`	enhanced
`Cat::FactorialDomain`	changed
`Cat::FiniteCollectionCat`	`Cat::FiniteCollection`
`Cat::HomogeneousFinite-CollectionCat`	`Cat::HomogeneousFiniteCollection`
`Cat::HomogeneousFinite-ProductCat`	`Cat::HomogeneousFiniteProduct`
`Cat::MatrixCat`	`Cat::Matrix`, changed
`Cat::PolynomialCat`	`Cat::Polynomial`
`Cat::SetCat`	`Cat::BaseCategory`, changed
`Cat::SquareMatrixCat`	`Cat::SquareMatrix`
`Cat::UnivariatePolynomialCat`	`Cat::UnivariatePolynomial`
`DIGITS`	changed
`Dom::AlgebraicExtension`	enhanced
`Dom::BaseDomain`	enhanced
`Dom::DistributedPolynomial`	enhanced, changed
`Dom::ExpressionField`	enhanced, changed
`Dom::Fraction`	enhanced
`Dom::Interval`	changed

Name in 1.4	New Name & Changed Features
Dom::Matrix	enhanced
Dom::MatrixGroup	enhanced, changed
Dom::Multiset	enhanced, changed
Dom::Pade	pade, enhanced, changed
Dom::Polynomial	enhanced, changed
Dom::Product	enhanced
Dom::SquareMatrix	enhanced, changed
Dpoly	polylib::Dpoly
EVAL_STMT	removed
Factor	removed, use factor instead
HISTORY	changed
Im	enhanced
LIB_PATH	LIBPATH
Line-Editor	enhanced
Network::AddEdges	Network::addEdge
Network::AddVertex	Network::addVertex
Network::AdmissibleFlow	Network::admissibleFlow
Network::AllShortPath	Network::allShortPath
Network::ChangeEdge	Network::changeEdge
Network::ChangeVertex	Network::changeVertex
Network::Complete	Network::complete
Network::ConvertSSQ	Network::convertSSQ
Network::Cycle	Network::cycle
Network::DelEdge	Network::delEdge
Network::DelVertex	Network::delVertex
Network::ECapacity	Network::eCapacity
Network::EWeight	Network::eWeight
Network::Edge	Network::edge
Network::Epost	Network::epost
Network::Epre	Network::epre
Network::InDegree	Network::inDegree
Network::IsEdge	Network::isEdge
Network::IsVertex	Network::isVertex
Network::LongPath	Network::longPath
Network::MaxFlow	Network::maxFlow
Network::MinCost	Network::minCost
Network::MinCut	Network::minCut
Network::OutDegree	Network::outDegree
Network::PrintGraph	Network::printGraph
Network::Random	Network::random
Network::ResidualNetwork	Network::residualNetwork
Network::ShortPath	Network::shortPath
Network::ShortPathTo	Network::shortPathTo
Network::ShowGraph	Network::showGraph

Name in 1.4	New Name & Changed Features
Network::TopSort	Network::topSort
Network::VWeight	Network::vWeight
Network::Vertex	Network::vertex
NIL	enhanced
NUMERIC	Type::Numeric
Poly	polylib::Poly
PRETTY_PRINT	PRETTYPRINT
Pref::floatFormat	changed
Pref::keepOrder	enhanced
Pref::printTimesDot	Pref::timesDot, changed
Pref::report	changed
PRINTLEVEL	removed
READ_PATH	READPATH
Re	enhanced
RootOf	changed
TEST_PATH	TESTPATH
UNIX	enhanced, changed
WRITE_PATH	WRITEPATH
alias	changed
anames	enhanced, changed
acos	arccos, enhanced
acosh	arccosh, enhanced
acot	arccot, enhanced
acoth	arccoth, enhanced
acsc	arccsc, enhanced
acsch	arccsch, enhanced
args	changed
array	enhanced, changed
asec	arcsec, enhanced
asech	arcsech, enhanced
asin	arcsin, enhanced
asinh	arcsinh, enhanced
assume	enhanced, changed
asympt	enhanced, changed
atan	arctan, arg, enhanced
atanh	arctanh, enhanced
assign_elems	assignElements
bernoulli	enhanced
besselK	enhanced
besselY	enhanced
beta	changed
binomial	changed
breakmap	misc::breakmap
built_in	builtin

Name in 1.4	New Name & Changed Features
`case`	enhanced
`catalan`	`CATALAN`
`changevar`	`intlib::changevar`
`combinat::bell`	changed
`combinat::permute`	changed
`combinat::powerset`	enhanced
`context`	changed
`contfrac`	`numlib::contfrac`
`cos`	changed
`cosh`	enhanced, changed
`cot`	changed
`coth`	enhanced, changed
`csc`	changed
`csch`	enhanced, changed
`debug`	changed
`decompose`	`polylib::decompose`
`degreevec`	enhanced
`diff`	changed
`dilog`	enhanced
`dirac`	enhanced
`discont`	enhanced
`discrim`	`polylib::discrim`
`domain`	`newDomain`
`domattr`	removed, use `slot` instead
`eint`	`Ei`
`erf`	enhanced
`erfc`	enhanced
`eval`	enhanced
`exp`	enhanced, changed
`expr`	changed
`extnops`	enhanced
`extop`	changed
`factor`	changed
`fft`	`numeric::fft`, enhanced
`for`	enhanced
`fread`	enhanced
`fun`, `func`	removed, use `->` instead
`func_env`	`funcenv`
`funcattr`	removed, use `slot` instead
`funcattr(sum, "float")`	`numeric::sum`, enhanced
`hastype`	enhanced
`heaviside`	enhanced
`history`	enhanced
`if`	enhanced

Name in 1.4	New Name & Changed Features
ifactor	changed
ifft	numeric::invfft, enhanced
igamma	enhanced, changed
igcd	changed
ilcm	changed
index_val	indexval
insert_ordered	listlib::insert, changed
int	enhanced, changed
io::readdata	import::readdata
io::readlisp	import::readlisp, changed
is	enhanced
lagrange	numeric::lagrange, enhanced, changed
last	changed
lcoeff	enhanced, changed
length	enhanced
level	changed
linalg::curl	changed
linalg::charMatrix	linalg::charmat
linalg::charPolynomial	linalg::charpoly
linalg::cholesky	linalg::factorCholesky, enhanced
linalg::delCol	changed
linalg::delRow	changed
linalg::det	changed
linalg::dimen	linalg::matdim
linalg::divergence	changed
linalg::eigenValues	linalg::eigenvalues, changed
linalg::eigenVectors	linalg::eigenvectors, changed
linalg::expr2Matrix	changed
linalg::extractMatrix	linalg::submatrix
linalg::factorQR	enhanced
linalg::grad	changed
linalg::hermiteForm	enhanced
linalg::isHermitian	linalg::isHermitean
linalg::isOrthogonal	linalg::isUnitary
linalg::linearSolve	linalg::matlinsolve
linalg::linearSolveLU	linalg::matlinsolveLU
linalg::nullSpace	linalg::nullspace
linalg::ogSystem	linalg::orthog
linalg::onSystem	removed, use linalg::orthog instead (in conjunction with linalg::normalize)
linalg::randomMatrix	enhanced
linalg::vectorDimen	linalg::vecdim

Name in 1.4	New Name & Changed Features
`linalg::vectorPotential`	changed
`linopt::maximize`	enhanced, changed
`linopt::minimize`	enhanced, changed
`linsert`	`listlib::insertAt`
`linsolve`	enhanced
`listtools::merge`	`listlib::merge`
`listtools::removeDupSorted`	`listlib::removeDupSorted`
`listtools::removeDuplicates`	`listlib::removeDuplicates`
`listtools::setDifference`	`listlib::setDifference`
`listtools::singleMerge`	`listlib::singleMerge`
`listtools::sublist`	`listlib::sublist`
`lmonomial`	enhanced, changed
`ln`	changed
`loadlib`	use `package` instead
`lterm`	enhanced, changed
`maprec`	`misc::maprec`
`misc::freeze`	`freeze`
`misc::genassop`	changed
`misc::makelib`	`prog::makeBinLib`
`misc::tableForm`	`output::tableForm`, enhanced
`misc::test`	`prog::test`, enhanced
`misc::unfreeze`	`unfreeze`
`nthcoeff`	enhanced, changed
`nthmonomial`	enhanced, changed
`nthterm`	enhanced, changed
`numeric::butcher`	enhanced
`numeric::det`	changed
`numeric::eigenvalues`	changed
`numeric::eigenvectors`	changed
`numeric::factorCholesky`	enhanced, changed
`numeric::factorLU`	enhanced, changed
`numeric::factorQR`	enhanced, changed
`numeric::fint`	`numeric::int`, enhanced, changed
`numeric::fsolve`	`numeric::realroots`, enhanced, changed
`numeric::gldata`	enhanced
`numeric::inverse`	changed
`numeric::minpoly`	`polylib::minpoly`
`numeric::odesolve`	enhanced
`numeric::quadrature`	enhanced
`numeric::singularvalues`	changed
`numeric::singularvectors`	changed
`numeric::vonMises`	`numeric::spectralradius`
`numlib::mpqs`	changed

Name in 1.4	New Name & Changed Features
optimize	generate::optimize
pdioe	solvelib::pdioe
phi	numlib::phi
plot2d	enhanced
plot3d	enhanced
plotOptions2d	enhanced
plotOptions3d	enhanced
plotfunc	plotfunc2d, plotfunc3d, enhanced, changed
plotlib::contourplot	plot::contour, changed
plotlib::cylindricalplot	plot::cylindrical, changed
plotlib::dataplot	plot::data, enhanced, changed
plotlib::densityplot	plot::density, changed
plotlib::polarplot	plot::polar, changed
plotlib::sphericalplot	plot::spherical, changed
plotlib::fieldplot	plot::vectorfield, changed
plotlib::xrotate	plot::xrotate, changed
plotlib::yrotate	plot::yrotate, changed
point	enhanced
primpart	polylib::primpart
print	enhanced
proc	enhanced
profile	prog::profile
protocol	enhanced
psi	enhanced
randpoly	polylib::randpoly
rationalize	changed
read	enhanced
rec	enhanced, changed
rectform	changed
repcom	@@, _fnest, enhanced
repeat	enhanced
resultant	polylib::resultant
rewrite	enhanced
sec	changed
sech	enhanced, changed
seq	removed, use _seqgen instead
series	enhanced
setuserinfo	enhanced
sharelib::Lsys	plot::Lsys
sharelib::animate	removed
sharelib::iroots	solvelib::iroots, use solve with option Domain = Dom::Integer instead

Name in 1.4	New Name & Changed Features
`sharelib::plotsetup`	removed
`sharelib::rational`	`numeric::rationalize`, enhanced, changed
`sharelib::trace`	`prog::trace`, enhanced
`sharelib::turtle`	`plot::Turtle`
`sharelib::untrace`	`prog::untrace`
`sign`	changed
`sin`	changed
`sinh`	enhanced, changed
`solve`	enhanced, changed
`sort`	changed
`spline`	`numeric::cubicSpline`, enhanced
`sqrfree`	`polylib::sqrfree`, changed
`stats::Tdist`	enhanced
`stats::linReg`	enhanced
`stats::mean`	enhanced
`stats::median`	enhanced
`stats::normal`	changed
`stats::stdev`	enhanced
`stats::variance`	enhanced
`string::contains`	`stringlib::contains`
`string::format`	`stringlib::format`
`string::formatf`	`stringlib::formatf`
`string::pos`	`stringlib::pos`
`string::delete`	`stringlib::remove`
`string::subs`	`stringlib::subs`
`string::subsop`	`stringlib::subsop`
`strlen`	removed, use `length` instead
`strmatch`	changed
`subs`	changed
`subsex`	changed
`sysorder`	changed
`table`	changed
`tan`	changed
`tanh`	enhanced, changed
`taylor`	changed
`this`	`dom`
`trace`	removed, use `prog::trace` instead
`transform::ifourier`	`transform::invfourier`, enhanced
`transform::ilaplace`	`transform::invlaplace`, enhanced
`transform::laplace`	enhanced
`unapply`	`fp::expr_unapply`
`unassign`	removed, use `delete` instead
`unassume`	changed

Name in 1.4	New Name & Changed Features
`userinfo`	changed
`while`	enhanced
`zeta`	changed

5 The new MuPAD Language

Lexical scoping is introduced with version 2.0 of MuPAD. This causes a lot of semantical changes in the MuPAD language which a library programmer must be aware of. Because of the big impact, all MuPAD programs must be revised, so we took the chance to change some other language features, too, which had been on the wish-list for some time.
The language description is organized as follows:

- Lexical Scoping
- Procedures and their Environments
- Special Identifiers
- No Pure Functions Any Longer
- The `$`-Operator
- New Operator `in`
- Slots instead of `domattr`s and `funcattr`s
- Deleting Values
- User-Defined Operators
- Scope of Aliases and User-Defined Operators
- Domains, Categories, and Axioms
- No `subs` of Domains Any Longer
- Functions `eval` and `hold` Work as Expected
- Operands and Output of Sets
- History contains Inputs and Outputs
- Renamed Functions and Variables
- Conditional Compilation
- Grammar
- Parallel Statements
- Debugging
- Overview

5.1 Lexical Scoping

The most prominent change in the MuPAD language for version 2.0 is the introduction of lexical scoping instead of dynamic scoping. Consider the following example:

```
>> x := 3:
>> q := proc() begin x end_proc:
>> p := proc(x) begin q() end_proc:
>> p(1)
```

In MuPAD version 1.4 or earlier, the call `p(1)` returns `1`. In version 2.0, it returns `3`. Surprise!

The reason is that due to dynamic scoping in MuPAD 1.4, the global variable `x` in `q` refers to the value of `x` in the current calling context, i.e., to the formal parameter `x` of `p`. The formal parameter `x` of `p` hides the value of the global identifier `x`.

In MuPAD 2.0, the variable `x` in `q` refers to the lexical context of `q`, i.e., to the global identifier `x`. The value of the formal parameter `x` of `p` does not influence the call of `q`.

If `q` would have been defined inside of `p`, then the global variable `x` of `q` would refer to the formal parameter `x` of `p`. Thus:

```
>> x := 3:
>> p := proc(x) local q; begin
        q:= proc() begin x end_proc;
        q()
   end_proc:
>> p(1)
```

would yield `1` in MuPAD version 2.0 as in version 1.4.

Why the change to lexical scoping? Three potential problems are caused by dynamic scoping, which are cured by lexical scoping. Two of them are classical ones which also exist in other dynamically scoped languages, such as some earlier LISP dialects or Perl prior to version 5. The third, particularly nasty problem is caused by the special evaluation rules of MuPAD.

5.1.1 The Funarg-Problem

This problem occurs if a procedure is used as a parameter for another one which contains free variables which conflict with local variables of the procedure called.

An example: Suppose we had no function `map` and want to implement a procedure which maps a function to the elements of a list:

```
list_map := proc(l, f) local t, r; begin
    r:= [];
    for t in l do
        r := append(r, f(t))
    end_for
end_proc
```

In MuPAD 1.4, this works as expected on first sight:

```
>> list_map([1, 2, 3], isprime)

        [FALSE, TRUE, TRUE]
```

Now another user, who does not know about the internals of our `list_map`, wants to use it to map his procedure `foo` to lists:

```
foo := proc(x) begin
    if x = t then t^2 else x end_if
end_proc:
```

Here, the global `t` is used to feed a further parameter to the procedure. Now he tries with MuPAD 1.4:

```
>> t := 2:
>> list_map([1, 2, 3], foo)

        [1, 4, 9]
```

It just does not work. The reason is quite clear: The free variable `t` in `foo` is bound to the local variable `t` of `list_map` instead of `2`, and the local variable always has the value of the parameter `x`. But how should he know this without knowing the implementation of `list_map`?

With MuPAD 2.0, the result is as expected:

```
>> t := 2:
>> list_map([1, 2, 3], foo)

        [1, 4, 3]
```

Here, the global `t` in `foo` is bound to the global identifier `t` in the lexical scope and not to the local `t` of `list_map` in the dynamic scope. Consequently, all works as expected.

5.1.2 The Closure-Problem, `option escape`

This problem occurs if a procedure escapes its context, i.e., if a procedure is returned from another one which contains free variables referring to local variables of the defining procedure. An example:

```
fpower := proc(f, n) begin
    proc(x) begin f(x)^n end_proc
end_proc:
```

The intended functionality is, given f and n, to return the function $x \to f(x)^n$. The call `fpower(sin, 3)`, for example, should return a procedure calculating `sin^3`.

It is clear that with dynamic scoping this cannot work the way shown above. With MuPAD 1.4, one gets:

```
>> sin3 := fpower(sin, 3):
>> f := cos: n := 2:
>> sin3(x)

        cos(x)^2
```

The free variables `f` and `n` of the returned procedure `sin3` are dynamically bound when `sin3` is executed and *not* when `sin3` is created (as was intended).

With lexical scoping in MuPAD 2.0, the outcome is as intended, provided that the procedure `fpower` is changed slightly:

```
fpower := proc(f, n) option escape; begin
    proc(x) begin f(x)^n end_proc
end_proc
```

The new procedure option `escape` states that a procedure may be returned which refers to the lexical context of the "enclosing" procedure. One must not forget to define this option, otherwise strange things will happen when the escaping procedure is executed. Now `fpower` works as expected:

```
>> sin3 := fpower(sin, 3):
>> f := cos: n := 2:
>> sin3(x)

        sin(x)^3
```

Here `f` and `n` are bound to the context which is current when the procedure `sin3` is created, i.e., to `sin` and `3`.

One can create functions like `fpower` even in MuPAD 1.4 by substituting the values of `f` and `n` into the procedure returned, but the code of `fpower` becomes ugly to read, error-prone and inefficient.

5.1.3 The LEVEL-Problem

This is the main source of trouble and confusion with lexical scoping. It occurs if identifiers are to be evaluated with a level greater than 1.

Inside of procedures, identifiers are usually evaluated with level 1, but by using the functions `eval` or `level` this may be changed—and sometimes must be changed in order to cause the simplification of expressions. Note that `level` always evaluates local variables in MuPAD 2.0 with level 1, as described in the next section.

During such evaluations, the values of local variables may conflict with global identifiers as shown in the following example:

```
f := proc(y) local x, LEVEL; begin
   x := y;
   LEVEL := 2;
   eval(x);
end_proc:
```

Here x is evaluated with level 2. Now with MuPAD 1.4:

```
>> unassign(x):
>> f(x^2 + 1)

        2     2
      (x  + 1)  + 1
```

What one would expect is the result x^2+1 because x has no value and thus the argument of f should not be changed further. What happens instead is that the expression x^2+1 is assigned to the local variable x. Then, with level 2, the local variable x is first evaluated to x^2+1 and in a second step to (x^2+1)^2+1. The local variable is mixed up with the free identifier x.

This problem is solved in MuPAD 2.0:

```
>> f := proc(y) local x; save LEVEL; begin
      x := y;
      LEVEL := 2;
      eval(x);
   end_proc:
>> delete x:
>> f(x^2 + 1)

          2
         x  + 1
```

Here the local variable x is evaluated to the expression x^2+1, where x is the global identifier x. This has no value such that the evaluation is stopped and x^2+1 is returned. (Also note that the function unassign has been changed to a delete-statement in version 2.0.)

5.1.4 Symbols and Variables

In order to implement lexical scoping, some new data types have been implemented in MuPAD 2.0. The most prominent is the date type DOM_VAR for local *variables* and formal parameters of procedures. Local variables and formal parameters are no longer identifiers of type DOM_IDENT but have this new type.

The best way is to think of DOM_VARs as programmatic variables. They always *must* have a value and are evaluated by simply being replaced by their value. Their evaluation is not influenced by LEVEL or level—this is a concept strictly restricted to identifiers. A warning is issued if a variable is used without being initialized. A variable which has not been initialized gets the value NIL.

Identifiers (DOM_IDENTs) are "names" which are lexically global and not bound by any local variables or formal parameters. They are evaluated as in previous MuPAD versions. (This also holds for procedures where identifiers are usually evaluated with level 1.) Personally, I prefer to think of identifiers as mathematical symbols which *may* have a value bound to them.

Because identifiers are no longer used as local variables or formal parameters, the concept of *environment variables* has undergone a significant change: If a local variable named LEVEL is used in MuPAD 2.0, it is not related in any way to the global identifier LEVEL used to control the evaluation level of identifiers. (In version 1.4, a local variable which had the same name as a "special identifier" was initialized with the current value of that identifier.) Nevertheless, most special identifiers such as LEVEL and DIGITS retain their meaning.

In order to make the temporary change of global identifiers easier, the new concept of *saved identifiers* is introduced with MuPAD 2.0. An example:

```
foo := proc(x, d) save DIGITS; begin
    DIGITS := d;
    float(x)
end_proc:
```

When foo is executed, the current value of the global identifier DIGITS is saved. DIGITS is then changed to float-evaluate x to the precision given by d. When foo is exited, the value of DIGITS is restored to the value saved on entry. This works no matter how the saving procedure is exited, may it be via return, error or some other means.

If a free symbol is needed in a procedure, one must either use a (global) identifier or create a new one with genident. One technique to get a free unbound symbol is the following:

```
foo := proc() save x; begin
    delete x;
    // use symbol x
    ...
end_proc:
```

Here x must not be bound by some "lexically outer" procedure, because the "use symbol x" part would otherwise refer to the lexically enclosing x, even though save x does refer to the global identifier x. By saving x, one ensures that the execution of foo does not destroy any value assigned to x. By deleting it, one gets a free symbol x which may then be used as polynomial indeterminate or free parameter, for example.

Note that properties are also saved and restored with save.

Note: Variables usually *must* be evaluated in the context (i.e., procedure) where they are defined. Variables may escape their context by using the option hold or the function hold, but this will almost *never* be intended. If one uses the option hold for a procedure, one should take care to evaluate the arguments by using the function context. Variables evaluated by context are evaluated in the calling procedure, where they have been defined. If they are evaluated by some other means (for example, eval or level), they are evaluated in the wrong procedure context—which is calling for trouble.

Marginal Note: Variables should usually not be created or manipulated "by hand", but by the MuPAD language parser. They have two operands which are non-negative

integer indices addressing their values relative to the procedure environment where they are defined. The first index is the "lexical distance" to the procedure environment where the variables value is contained in; this index is 0 if the variable is defined in the procedure where it is used. The second index is the position of the variables value in its procedure environment; here the first local variable has index 2 and the actual parameters follow the local variables. (The special variables `procname` and `dom` have index 0 and 1.)

One may create variables by calling `DOM_VAR(i, j)`, where `i` and `j` are non-negative integers. •

5.1.5 Accessing Arguments with `args`

Please note that the function `args` has changed in a quite subtle way: In MuPAD versions before 2.0, `args(i)` returned the `i`th actual parameter of a procedure call. In version 2.0, `args(i)` returns the actual value of the variable holding the `i`th parameter. Thus the value of `args(i)` changes when the variable holding the parameter is changed. This was not the case in former MuPAD versions. E.g.,

```
>> f := proc(x) begin x := PI; args(1) end_proc:
>> f(1)
```

produced the result `1` in MuPAD 1.4, whereas MuPAD 2.0 produces `PI`.

5.2 Procedures and their Environments

In addition to the introduction of lexical scoping and saved variables, three new features have been introduced for procedures. Arguments may get default values, the domain a procedure belongs to may be accessed, and types of local variables may be declared.

5.2.1 Default Argument Values

Default values may be defined for arguments which are not supplied by the caller at run-time. The syntax for formal procedure parameters is now *name* `=` *default-value* `:` *type*, where the default value and the type declaration may be missing:

```
>> foo := proc(p = 0, v = x : DOM_IDENT) : DOM_POLY
   begin
       poly(p, [v], Expr)
   end_proc:
>> foo(), foo(x+y+1), foo(x+y+1, y)

   poly(0, [x]), poly(x + (y + 1), [x]), poly(y + (x + 1), [y])
```

The actual parameters are assigned to the formal ones in the corresponding order. If there are formal parameters which are not supplied by actual ones, the default values are used for them. If no default value was defined, `NIL` is used as actual value and a warning is printed. If a type but no default value is declared for a formal parameter and that parameter is not supplied by an

actual value, a run-time error results (but only if parameter type checking is enabled for the corresponding procedure call).

The default values and type expressions for the parameters and return value belong to the lexically enclosing context and *not* to the procedure at hand. This is also the context where they are bound:

```
>> otto := proc(x) option escape; begin
       proc(x = x) begin x end_proc
   end_proc:
>> otto13 := otto(13):
>> otto13(), otto13(14)

        13, 14
```

Here the default value `x` is bound to `otto` and not to the procedure returned by `otto`. Note that the option `escape` must be defined here because the escaping procedure refers to the argument `x` of the context of the outer procedure `otto`.

The default values, argument types, and return type of a procedure are evaluated when the procedure definition is evaluated and *not* when the procedure is executed. (In version 1.4, the types where evaluated each time the procedure was executed.) This is shown by the following example:

```
>> VAL := 3: TYP := DOM_INT:
>> otto := proc(x = VAL : TYP) : TYP begin x end:
>> otto()

        3

>> VAL := 10: TYP := DOM_STRING:
>> otto()

        3
```

The procedure `otto` does not depend on the current values of the identifiers `VAL` and `TYP`.

5.2.2 Accessing the Domain a Procedure Belongs To

If a procedure is an entry of a domain, this domain may be accessed inside the procedure via the special variable `dom`: When a procedure is inserted into a domain, the domain is stored in the procedure definition. When the procedure is executed, the variable `dom` gets this domain as value; `dom` gets the value `NIL` if the procedure was not inserted into any domain.

The domain is only inserted into the procedure definition if the procedure is directly inserted into the domain. If, for example, the procedure is an element of a list which is inserted into a domain, the domain is *not* inserted into the procedure definition.

One may view `dom` and `procname` as local variables implicitly declared for any procedure: The domain a procedure belongs to is only accessible via `dom` inside the body of the procedure—not in any procedure called from or lexically enclosed in the procedure. (The same holds for the special variable `procname`.)

Note: The special variable `dom` replaces the place-holder `this` in the domains package. It has the advantage that it needs not to be substituted into the domain entries when they are created on the fly.

5.2.3 Variable Type Declarations

Types of local variables may now be declared, using the syntax *name* `:` *type*, as in `i : DOM_INT`. Types may be defined by arbitrary expressions.

Type declarations for local variables currently do *not* have any functionality—so this feature seems to be quite absurd. But it may be useful for future converters from MuPAD language to C or Fortran code, for future debugging tools, or right now for documentation.

If you declare variable types you should be serious about it because some day types may be checked or type information may be used in some other way.

5.2.4 Operands of Procedures

Procedures got five new operands in version 2.0, these are the operands 9 through 13:

9: An expression sequence containing the types of the local variables. If a type is not defined for a variable, the corresponding entry is `NIL`. If no type is defined, the operand is `NIL`.

10: An expression sequence containing the default values of the formal parameters. If a default is not defined for a parameter, the corresponding entry is `NIL`. If no defaults are defined, the operand is `NIL`.

11: An expression sequence containing the identifiers to be saved or `NIL` if no identifiers are saved.

12: The procedure environment of the procedure instance which was executed when the procedure instance was created.

13: The domain a procedure belongs to or `NIL` if the procedure has not been inserted into a domain.

One should not need to know the details about the 12th operand, thus you may skip the following section.

5.2.5 Procedure Environments

Marginal Note: In version 2.0, a distinction is made between a *procedure definition* and a *procedure instance*. A procedure definition is created by the parser or by the function `_procdef`. A procedure definition is evaluated to a procedure instance. A procedure

instance refers to the procedure environment which was active when the procedure definition was evaluated.

But what is a *procedure environment*? Each time a procedure instance is executed, a procedure environment is created for the instance. The environment contains the values of the parameters and local variables of a procedure instance during its execution. Variables refer to procedure environments. Procedure environments have the new data type `DOM_PROC_ENV`.

A procedure environment additionally contains the procedure executed, the name of this procedure, the domain the procedure belongs to, the procedure environment of the calling procedure instance, and the environment of the procedure which was executed when the current procedure instance was created.

This last entry allows a variable to refer to an lexically outer context, which just happens to be the environment of the procedure instance which was active when the actually executed procedure was instantiated.

Procedure definitions and instances are both of type `DOM_PROC`. A definition has the value `FAIL` as 12th operand, a procedure instance has a procedure environment or `NIL` as 12th operand. The procedure environment is the environment of the procedure instance which was executed when the procedure instance at hand was created. It is inserted into the procedure environment when the instance is executed. The "environment operand" is `NIL` if the procedure instance does not contain any variables referring to an outer lexical scope.

If the option `escape` is not defined for a procedure, the procedure's environment is deleted when the procedure execution has finished. Strange errors will occur if such a deleted procedure environment is referred to later on. On the other hand, if the option `escape` *is* defined, the procedure environment may still be referred to after the procedure has been executed—allowing escaping procedures to refer to variables stored in the environment.

Procedure environments should be regarded as opaque and you should not try to inspect them (for example, with `op`), leave alone manipulate them.

•

5.3 Special Identifiers (Environment Variables)

As was described above, the use of environment variables as local variables of procedures is no longer supported in MuPAD 2.0. One should use the `save` declaration to save and restore identifier values.

Nevertheless, most "special identifiers" like `DIGITS`, `TEXTWIDTH` or `LEVEL` retain their meaning in version 2.0. Only some seldom used identifiers have been removed: `PRINTLEVEL`, `ERRORLEVEL` and `EVAL_STMT` are no longer supported and have no special meaning any more. Most special identifiers lost the underscores in their names, so `PRETTY_PRINT` is now `PRETTYPRINT`, `READ_PATH` is `READPATH` and so on.

The meaning of `HISTORY` has been changed: It is no longer possible to change the length of the history table of procedures. The history table of a procedure can no longer be enlarged and always has three entries. Thus the notation `HISTORY := [i,j]` is no longer valid, only `HISTORY := i` with a non-negative integer `i` is allowed, controlling how many interactively entered commands are stored in the global history table. (The access to the global history table has also been changed, see below.)

The former special identifier `procname` now is a variable which implicitly is declared for any procedure, similar to the variable `dom`.

5.4 No Pure Functions Any Longer

The concept of pure functions has been abandoned with MuPAD 2.0. The functions `fun` and `func` no longer exist. (Also the internally used functions `newpurefunc` and `newfuncarg` have been removed.) Instead of `func`, the arrow operator `->` was introduced:

```
>> f := (x, y) -> x^2 * sin(y):
```

This operator creates a procedure:

```
>> f(PI, z)

        PI^2 sin(z)
```

Pure functions existed for efficiency reasons. Due to the changes in version 2.0, procedure execution is now as fast as the execution of pure functions used to be. This made pure functions obsolete.

Further, pure functions defined by `fun` were not easy to read because `args` had to be used to refer to the arguments. Argument references in pure functions defined by `func` were created by substituting calls to `args` into the defining expression—a quite slow and error-prone method to define a binding.

The consequence was to remove pure functions entirely. As a side-effect, the function `block` is of no use any longer and also has been removed.

5.5 The $-Operator

The syntax of the "sequence generator" `$` has been changed. Given the expression `f(i) $ i=1..10`, the identifier `i` was not allowed to hold a value in MuPAD 1.4. This was very cumbersome, one often had to write `f(i) $ hold(i)=1..10` in order to ensure that the index `i` was a "pure name".

This syntax has changed in order to resemble the `for`-loop: Now `i` *must* be an identifier or variable, which may have a value or not.

In the case `f(i) $ i=1..10` the `$`-operator is regarded as a ternary operator with operands `f(i)`, `i`, and `1..10`. The new rules for the `$`-operator are:

sequence-expression:
- *`$` relation*
- *relation `$` relation*
- *relation `$` name = relation*
- *relation `$` name `in` relation*

Note: It is no longer allowed to use expressions like `hold(i)` for the index variable definition. An expression like `f(i) $ hold(i)=1..10` is parsed as binary expression `_seqgen(f(i), hold(i)= 1..10)` and results in a run-time error.

5.6 New Operator `in`

A new binary operator `in` exists which may be used to test if an expression is member of a given set. The priority of the operator is the same as for other relations like = or >:

```
>> x in {3, 5}

        x = 3 or x = 5
```

5.7 Slots instead of `domattrs` and `funcattrs`

The functions `domattr` and `funcattr` have been unified to a single new concept, the *slot*. A slot is a named entry of a MuPAD datum which may hold a value.

One may access and change the value of a slot using the new function `slot`:

`slot(d, n)` returns the value of the slot named `n` of the datum `d`. `n` may be an arbitrary datum.

`slot(d, n, v)` changes the value of the slot named `n` of the datum `d` to `v`.

For most objects, a copy of the original datum `d` with changed slot is returned, the original value of `d` is not changed as a side-effect. The only exception are domains: their slots are changes as a side-effect. (This is due to the so-called "reference effect" of domains.)

What happens if a slot named `n` does not exist? This depends on the domain of the datum `d`:

- If the objects of a domain have open-ended slots, arbitrary new slots may be created by simply inserting values with new slot names. If a slot is accessed which does not yet exist, `FAIL` is returned as the "value" of the slot.
- If a non-existing slot is accessed for an object of a domain with fixed slots, an error occurs. This holds for read and write access.

Each datum has a special slot named `"dom"`. This is a read-only slot which holds the domain the datum belongs to. Thus `slot(d, "dom")` is equivalent to `domtype(d)`. The value of this slot cannot be changed.

Apart from this special slot, currently only two MuPAD domains have slots: domains and function environments. The entries of a domain or function environments are stored in slots and can be accessed and changed with the function `slot`. Both domains have open-ended slots: New slots may be created by simply inserting values with new slot names.

Thus `domattr(d, n)` becomes `slot(d, n)` if `d` is a domain. Similarly, `funcattr(f, n)` becomes `slot(f, n)` if `f` is a function environment.

Thus the functions `domattr` and `funcattr` are no longer needed and have been removed.

Note that `slot` is somewhat restricted compared to the former functions:

- If the first argument d in domattr(d, n) was no domain and the domain of d did not have a method "elemattr", then the domain entry n of the domain of d was returned. This is no longer the case with slot. Now the domain of d must be used explicitly, as in slot(domtype(d), n).
- If the first argument f in funcattr(f, n) was no function environment, then a function environment with executing function f was created implicitly by funcattr. This is no longer the case with slot. Now a function environment for f must be created explicitly, as in slot(funcenv(f), n). (Note that funcenv, which used to be the function func_env in version 1.4, now may be called giving the executing function only, the output function may be omitted.)

One may define slots for other domains by overloading the function slot.

For domains there is a special mechanism to create new slot values on demand: If a slot is read which does not yet exist, 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 as for other open-ended slots.) The method "make_slot" works exactly as the former domain method domattr, but has been renamed for obvious reasons.

5.7.1 Accessing Slots Using the ::-Operator

Similar to the former function domattr, the ::-operator is a shorthand 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"). Thus, in order to access a slot "n" of the domain of a datum d, one may simply write d::dom::n instead of slot(domtype(d), "n").

One may not only access but also change the value of a slot using the ::-operator. Similar to an indexed assignment, an assignment of the form d::n := v assigns the value v to the slot named "n" of d. (An equivalent syntax for such an assignment to a slot is slot(d, "n") := v.)

Please note that there is a subtle semantical difference between a slot assignment using the call d := slot(d, "n", v) and a slot assignment of the form d::n := v. The function slot evaluates its arguments as usual, but the left hand side of a slot assignment d::n := v is evaluated as for an indexed assignment a[b] := c. This means that d is not fully evaluated and then v assigned to the slot n of the result, rather the slot of the value of the variable or identifier d is changed and the result assigned to d.

This is different to the former function domattr: In version 1.4, the datum d on the left hand side of an assignment of the form d::n := v (which was equivalent to domattr(d, "n") := v) was fully evaluated and then the entry of the resulting domain was changed. (To complicate things further: If d did not evaluate to a domain the domain of the resulting datum was changed.) This was possible because domains show the so-called "reference effect": their entries were changed by the assignment as side-effect.

In order to get the same functionality as the former assignment to `domattr`, one may change a domain slot by a call of the form `slot(d, "n", v)`, which as a side-effect changes the domain `d` as described above.

At the bottom line, the new slot assignment is much more regular than the former assignment to domain entries. Furthermore, it is consistent with the indexed assignment.

5.7.2 Overloading the `slot` Function

By overloading the `slot` function, slot access and slot assignment can be implemented for other objects than domains or function environments. The domain method `"elemattr"` is no longer needed, overloading `slot` does a better job. One may even overload `slot` for basic domains other than domains and function environments.

The following function defines the fixed slots `"numer"` and `"denom"` for the rational numbers:

```
unprotect(DOM_RAT):
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") end;
      if args(0) = 3 then
          subsop(r, i = v)
      else
          op(r, i)
      end
  end_proc:
protect(DOM_RAT, Error)
```

5.7.3 No Keywords as Operands of `::`

In version 1.4, the second operand of the `::`-operator was allowed to be a keyword, like in `Z7::name`. This was quite crude. In version 2.0, keywords are no longer allowed, the second operand *must* have the syntax of an identifier.

The following two standard names for domain entries have been changed accordingly:

`name`	now called	`Name`,
`not`	now called	`_not`.

5.8 Deleting Values

Values of identifiers and variables may be deleted by using the `delete` statement instead of the former function `unassign`:

```
>> delete a, T[1]
```

Please refer to the grammar below for the exact syntax of the delete statement.

By using the "underline function" `_delete`, values can also be deleted in a functional manner.

5.9 User-Defined Operators

The user may now define new unary or binary operators. A user-defined operator may be a unary pre- or postfix operator or a binary infix operator. A binary infix operator may be associative or not. The priority of the operator may also be defined.

New operators are defined by using the function `operator`. The syntax is

`operator(`*name*`,` *function*`,` *type*`,` *priority*`)`

where *name* is a string with the operator token, *function* the operator of the function call created by the parser, *type* the type of the operator and *priority* its priority. The priority must be a number between 1 and 1999. The following operator types exist:

`Prefix`	unary prefix
`Postfix`	unary postfix
`Binary`	binary non-associative infix
`Nary`	binary associative infix

Thus given an operator defined by `operator("**", foo, Postfix, 500)`, the expression `x**` is parsed as `foo(x)`. With the definition `operator("**", foo, Binary, 500)` the expression `x**y**z` is parsed as `foo(foo(x, y), z)` (non-associative binary operators bind left-to-right). Given the definition `operator("**", foo, Nary, 500)`, the expression `x**y**z` is parsed as `foo(x, y, z)`.

The operator name may be up to 32 characters long, and it must not start with whitespace nor with a backslash (\). The function `operator` does not check these restrictions, operators thus defined simply won't work.

This new concept has made the &-operator and the function `bin_op` obsolete. Both have been removed from the language.

5.10 Scope of Aliases and User-Defined Operators

When reading a file, the scope of an alias-definition or user-defined operator depends on the way the file is read.

A new option `Plain` has been implemented for the function `read` which restricts the scope of aliases and user-defined operators: If this option is given, the file is read in a "fresh" parser context where no aliases and user-defined operators initially exist. Aliases and user-defined operators defined in the file read in are restricted to the scope of that file, they are not "exported" into the context where the file is read.

Additionally, there exists no history table when the file is read `Plain`, thus the history table of the reading context is not changed by reading the file.

The reason for introducing this option is that library files are read as side-effect during evaluation. Thus, if the scope of aliases and user-operators would not have been restricted, they could change the meaning of the files read in an uncontrolled way.

On the other hand, if the file is not read with option `Plain`, it is parsed in the context of the call of `read`. Thus aliases and user-operators take effect when executing the files commands and alias or user-operator definitions in the file are "exported" into the reading context. Additionally, the files commands are entered into the history table.

5.11 Domains, Categories and Axioms

New language constructs have been introduced with MuPAD 2.0 for domain-, category and axiom constructors. They replace awkward functions such as `DomainConstructor::new` used to define constructors in version 1.4. The syntax of the new constructor statements is described below. The former `new`-methods of the constructors no longer exist.

Even more important is the new implementation of the constructors. Constructor parameters and local values are now variables (`DOM_VAR`s) bound by their lexical scope. Entries defined inside domain- and category constructors access the constructor parameters and locals as if they were defined in an lexically outer procedure.

Constructor parameters and local values are no longer simply substituted into the domain entries. This means, for example, that parameters and locals of methods may have the same names and hide the constructors names—a source of nasty errors in version 1.4. Another consequence is that parameters and locals may be changed by domain methods—they are in fact variables which are local to the domain but global to the domain's methods (similar to static class members in C++).

The special name `this` has been replaced by the special variable `dom`, as has been described above.

Marginal Note: The actual domain is no longer inserted into a methods body by substituting the identifier `this`, instead `dom` is used to access the actual domain.

Constructor parameters and local values are now defined by the parser when the constructor is read. Each domain method is enclosed by a procedure environment which is unique for the domain or category which defined the method.

This means that domain entries are created much faster and need less memory, because the bodies of the methods need not be changed by substitutions any longer. •

Some examples are shown which explain the new syntax. The exact grammar is described below.

The simplest case is a constructor without parameters:

```
domain d
    Name := "simple_domain";
    info := "a simple example";
end_domain;
```

Here a domain with key `d` is created and assigned to the identifier `d` as a side-effect. Additionally, two entries named `Name` and `info` are defined for the domain.

Note: The former domain-creating function `domain` has been renamed because `domain` is a keyword now. The function is called `newDomain` now.

Apart from the renaming of the function `domain`, no other changes have been introduced for domains (data type `DOM_DOMAIN`). Only domain constructors are affected by the changes.

A mathematically more meaningful domain has a super-domain it inherits, categories it belongs to, and axioms which are stated:

```
domain Dom::Integer
    inherits Dom::Numerical;
    category Cat::EuclideanDomain, Cat::FactorialDomain, ...;
    axiom Ax::canonicalRep, Ax::systemRep, ...;

    info_str := "domain of integer numbers";
    testtype := i -> if domtype(i) = DOM_INT then TRUE
                     else FAIL end_if;
    ...
end_domain;
```

Here the domain with key `Dom::Integer` is defined and assigned to `Dom::Integer`. It inherits the entries of the domain `Dom::Numerical` (which is its direct super-domain) and belongs to the categories `Cat::EuclideanDomain` etc. Further, the axioms `Ax::canonicalRep` etc. are assumed. The first entry defined for the domain is `info_str`.

A domain may inherit from only one single direct super-domain. Statements defining the super-domain, categories and axioms may be given in any order, multiple category or axiom statements are allowed. The entries must follow these statements.

A domain constructor may have formal parameters and local variables:

```
domain Dom::IntegerMod(Mod: Type::PosInt)
    local zero;
    inherits Dom::BaseDomain;
    category if isprime(Mod) then Cat::Field
             else Cat::CommutativeRing end_if;
    axiom Ax::canonicalRep, ...;

    // entries:
    characteristic := Mod;
    size := Mod;
    ...

begin
    if Mod < 2 then error("modulus must be > 1") end;
    zero := new(dom, 0);
end_domain;
```

Here the domain constructor `Dom::IntegerMod` with parameter `Mod` and local variable `zero` is defined. Parameters may defined as for procedures, default values and type declarations are allowed. The types of local variables may also be defined.

Please note that local variables must be declared first, in front of the other declarations. Like in procedures, the default values and type declarations are parsed in the outer lexical context, whereas any other definitions are parsed in the lexical context of the constructor.

When a domain is created by the constructor, the actual parameters are assigned to the formal ones, then the statements between `begin` and `end_domain` are executed. This defines the context where the domain entries are evaluated. Thus an entry like the category definition `if isprime(Mod) then Cat::Field ...` above is evaluated in a context where `Mod` has the value of the actual parameter of the domain constructor. If `Mod` is prime, the domain gets the category `Cat::Field`, otherwise `Cat::CommutativeRing`.

The parameters and local variables of a domain context may be changed in the methods of the domain. One may regard these values as variables which are global to the domain entries but local to the domain.

The syntax for category constructors is very similar to that for domains. Only the `inherits`-statement must not exist:

```
category Cat::UnivariateSkewPolynomial(R: DOM_DOMAIN)
    local hasField;
    category Cat::Ring, Cat::LeftModule(R), ...;
    axiom Ax::normalRep, ...;

    // entries:
    _plus; ore_mult; _mult; ...
    ...

begin
    hasField := bool(R::hasProp(Cat::Field));
end_category;
```

The definition of an entry may be missing in a category constructor. This means that this is a required entry which *must* be defined in a sub-category- or domain constructor.

Axioms are much easier. They allow only parameters and local variables, but no other attributes:

```
axiom Ax::efficientOperation(oper: DOM_STRING)
begin
    if args(0) <> 1 then error("wrong no of args") end_if;
end_axiom;
```

5.12 No subs of Domains Any Longer

The substitution functions `subs` and `subsex` do no longer change do-

mains, they are simply ignored.

In earlier versions, subs and subsex changed domains as *side-effect*, as in:

```
>> d := domain("d"):
>> d::a := b:
>> d2 := subs(d, b = 13):
>> d::a

        13
```

In most cases, this behavior was undesired, causing confusion. Now domains are no longer touched by subs and subsex.

5.13 Functions eval and hold Work as Expected

In version 2.0, the function eval works for any argument: It evaluates its argument as usual and then evaluates the result again. The result of the second evaluation is returned.

Marginal Note: In former versions eval did only evaluate twice if the argument was an expression of a certain type, for example, if it was a call of last or subs. This was a constant source of confusion. Now eval works as expected and evaluates any argument twice. •

The function hold has also been changed: Now hold simply returns its arguments without evaluating them.

Marginal Note: This was true in former versions only if hold was not called inside hold. If hold was called inside of hold, the inner calls could disappear under certain circumstances. It was not true that a call of eval would cancel a call of hold in any case. •

5.14 Operands and Output of Sets

The ordering of elements of finite sets (kernel domain DOM_SET) depends on the insertion ordering of the elements. This was always the case, but is much more obvious now due to a changed implementation of sets. Thus two equal sets (equal in the sense of the system function _equal) may have a different ordering of their elements.

This seems to be quite confusing to most users. In order to avoid this confusion, the elements of sets are now sorted *for output and indexed access*. (The ordering used for sorting the elements is the one given by sysorder.) When accessing the elements with the function op, the internal ordering of the set elements is used.

The following sets A and B have the same output and are considered equal by _equal:

```
>> A := {a}: A := A union {b}

        {a, b}

>> B := {b}: B:= B union {a}
```

```
        {a, b}

>> bool(A=B)

        TRUE
```

Indexed access yields the same elements:

```
>> A[i] $ i=1..2

        a, b

>> B[i] $ i=1..2

        a, b
```

Only `op` reveals that the "internal" ordering of the sets is different:

```
>> op(A)

        a, b

>> op(B)

        b, a
```

One may change the ordering of sets for output and indexed access by changing the slot `"sort"` of `DOM_SET`.

One should bear in mind:

- The ordering of set elements for output and indexed access is different from the ordering obtained by `op`.
- Indexed access is much slower than access with `op` because the set elements are sorted first.

5.15 History contains Inputs and Outputs

Since version 2.0, both the inputs and the results of interactively entered statements are stored in the global history table. A call of `history()` no longer prints out the history table as a side-effect but returns the number of commands entered interactively. A call of `history(i)` returns a list containing the *i*-th input and result if that command still is contained in the history table. (The *i*-th command is removed from the history table if *i* is less than `history() - HISTORY`.)

Please note that only the final result of an interactively entered statement is entered into the history table. In former versions of MuPAD, any results of statements which where executed at the interactive level where entered into

the history table. The following for-statement, for example, entered the values 2, 3 and 4 into the history table. Now only the result of the whole for-statement, which is 4, is entered into the history table:

```
>> for i from 1 to 3 do i+1 end_for:
>> %

        4
```

One may assign 0 to `HISTORY`, which disables the insertion of commands into the history table.

Commands executed when reading a file are also entered into the history table. This may be suppressed by using the new option `Plain` of the function `read`.

5.16 Renamed Functions and Variables

Several kernel functions and special variables have been renamed because the former names did not adhere to the naming conventions.

Old Name	New Name
`PRETTY_PRINT`	`PRETTYPRINT`
`READ_PATH`	`READPATH`
`WRITE_PATH`	`WRITEPATH`
`LIB_PATH`	`LIBPATH`

Table 5.1: Renamed Special Identifiers

Old Name	New Name
`assign_elems`	`assignElements`
`built_in`	`builtin`
`domain`	`newDomain`
`domattr`	`slot`
`func_env`	`funcenv`
`funcattr`	`slot`
`index_val`	`indexval`

Table 5.2: Renamed Kernel Functions

5.17 Conditional Compilation

MuPAD 2.0 has a new type of control statement, called `%if`. It controls the generation of code by the parser, that is, with `%if` you can exclude statements from ever reaching the internal form of your program. The syntax of an `%if` statement is the same as of an `if` statement, only the keyword `if` must be replaced by the string `%if`:

pre-if-statement:

`%if` condition `then` statement-seq elif-seq$_{opt}$ *else-part*$_{opt}$ *end-if*

elif-seq:
elif-part
elif-part elif-seq

elif-part:
`elif` *condition* `then` *statement-seq*

else-part:
`else` *statement-seq*

end-if:
`end_if`
`end`

The semantics is as follows:

- First, the condition following the `%if` is parsed and evaluated directly by the parser. This evaluation must return a Boolean value; otherwise, an error is raised.
- If the result is `TRUE`, the then-part (i.e., the statement sequence following the keyword `then`) is returned by the parser. The remaining parts of the `%if` statement are ignored by the parser.
- If the condition does not evaluate to `TRUE`, the then-part is ignored. If an `elif`-part exists, the condition of that part is evaluated.
- If that condition returns `TRUE`, the corresponding then-part is returned by the parser and the rest of the `%if` statement is ignored.
- If the condition does not evaluate to `TRUE`, the then-part is ignored and the remaining parts are parsed as described before.
- Finally, if no condition evaluates to `TRUE`, the else-part is returned by the parser. If no else-part exists, `NIL` is returned.

Note that even if a part of the `%if` statement is ignored, it must be syntactically correct. Thus the conditional compilation is not handled on the lexical level, rather on the grammar level of the language.

The conditions are parsed in their lexical context, but are evaluated by the parser in the context where the parser is executed. Therefore one must *not* refer to local variables from the enclosing lexical context in the conditions. The parser does not check this.

5.18 Grammar

A new parser had to be implemented for lexical scoping. This opportunity was taken to change the grammar in several ways to make it somewhat more user-friendly and regular.

Most of the changes are compatible to the previous version 1.4. The most serious incompatibility is the change of the syntax of the `$`-operator as described above.

Other points of interest are:

- Interactively entered statements need no longer be closed by a separator (`;` or `:`).
- Functions may be defined by the arrow-operator `->`.
- Program control statements and procedure definitions may be optionally followed by `end` instead of `end_if`, `end_while` and so on.
- Program control statements are grammatically handled like names or literals. This means one can enter statements like `x := if x >= 0 then x else -x end;` without needing to enclose the if-statement into brackets.
- Expressions like `a := b := c` and `a::b::c` are now syntactically valid.
- One may use arbitrary strings as identifier and variable names by quoting them with ‘ (back-quote character), such as `‘&uuml;‘` or `‘+‘`. (The latter is predefined to the function `_plus`.) Two consecutive back-quotes must be used to insert a back-quote into the name, as in `‘Tom‘‘s back-quote‘`.

5.18.1 Statements

A legal MuPAD input entered interactively or read from a file must be a *statement-seq*:

statement-seq:
statement$_{opt}$ *separator statement-seq*
statement$_{opt}$ *separator*$_{opt}$

separator:
`;`
`:`

The last statement entered interactively or read from a file need not be closed by a separator `;` or `:` any longer. (If the last separator is missing, the output of the last statement is printed if entered interactively.) In addition to statements, help and system commands may be entered during interactive input as before.

statement:
expression-seq
expression `:=` *statement*
`delete` *expression-seq*$_{opt}$

expression-seq:
function
function `,` *expression-seq*

function:
expression
name `->` *function*
`(` *name-seq*$_{opt}$ `)` `->` *function*

name-seq:
name
name `,` *name-seq*

Please note that the left hand side of assignments are of course not arbitrary expressions. The correctness of left hand sides if decided partly by attributes computed by the parser and partly by the `_assign` function at run-time. The former grammar had a problem here, it was not $LL(n)$ for any n.

5.18.2 Expressions

The basic rules for expressions are the following:
expression:
factor
sequence-expression
prefix-operator expression
expression infix-operator expression
expression postfix-operator
expression `(` *expression-seq*$_{opt}$ `)`
expression `[` *expression-seq*$_{opt}$ `]`

sequence-expression:
`$` *expression*
expression `$` *expression*
expression `$` *name* `=` *expression*
expression `$` *name* `in` *expression*

The actual implementation uses operator precedences to parse expressions. The predefined operators are shown in table 5.3. The operator type "n-ary" means an associative binary operator. Function call and indexed access may also be considered as operators and thus also have a priority.

The higher the precedence, the more "tightly" the operators bind their arguments, so that for example `1+2*3=7`.

Token	Type	Priority
`or`	n-ary	100
`and`	n-ary	200
`not`	prefix	300
`$`	prefix, binary, 3-nary	300
`=`, `<`, `<=`, `>`, `>=`, `<>`, `in`	binary	400
`..`	binary	500
`union`	n-ary	600
`minus`	binary	700
`intersect`	n-ary	800
`mod`, `div`	binary	900
`+`	prefix, n-ary	1000
`-`	prefix, binary	1000
`*`	n-ary	1100
`/`	binary	1100
`^`	binary	1200
`!`	postfix	1300
`@`	n-ary	1500
`@@`	binary	1600
`.`	n-ary	1700
f`(`x`)`, a`[`i`]`	*see grammar*	1800
`'`	postfix	1900
`::`	*see grammar*	2000

Table 5.3: Predefined Operators

The precedences are incremented by 100 to allow for user-defined operators which "fit between" predefined ones. The user may additionally define his own pre-, post- and infix operators, these must have a precedence between 1 and 1999.

5.18.3 Factors

Control statements are now factors instead of top-level statements:

factor:
`(` *statement-seq* `)` *domain-entry*$_{opt}$
control-statement
procedure-definition
domain-constructor
category-constructor
axiom-constructor
`[` *expression-seq*$_{opt}$ `]`
`{` *expression-seq*$_{opt}$ `}`
name domain-entry$_{opt}$
% integer
literal

domain-entry:
`::` *name*
`::` *name domain-entry*

The syntax of the control statements containing statement sequences (if-, case-, for-, while- and repeat statement) has been extended with respect to the closing keywords. One may now also optionally use simply `end` instead of `end_if`, `end_case`, `end_for`, `end_while` and `end_repeat` to close these statements.

There is a new control statement for conditional code inclusion. It has the same syntax as an `if`-statement, only the keyword `if` is replaced by the token `%if`.

The syntax of the "atomic" control statements (`next` and `break` statements) and literals like names and numbers has not been changed. The syntax for names has been extended by additionally allowing arbitrary strings.

5.18.4 Procedures

Several new features have been added to procedure definitions as described above. The grammar rules have been extended accordingly:

procedure-definition:
`proc (` *argument-seq*$_{opt}$ `)` *type*$_{opt}$
declaration-seq$_{opt}$
`begin` *statement-seq end-proc*

argument-seq:

formal-argument
formal-argument `,` *argument-seq*

formal-argument:
name default-value$_{opt}$ *type*$_{opt}$

default-value:
`=` *expression*

type:
`:` *expression*

declaration-seq:
declaration `;`
declaration `;` *declaration-seq*

declaration:
`name` *expression*
`local` *local-var-seq*
`save` *name-seq*
`option` *option-seq*

local-var-seq:
local-var
local-var `,` *local-var-seq*

local-var:
name type$_{opt}$

option-seq:
option
option `,` *option-seq*

option:
`hold`
`remember`
`escape`
`arrow`
`noDebug`

end-proc:
`end`
`end_proc`

5.18.5 Domains, Categories, and Axioms

New language constructs have been defined for domain-, category- and ax-

iom constructors. Constructor parameters and local variables are now bound lexically, their scope is the constructor definition:

domain-constructor:
`domain` *factor local-seq*$_{opt}$ *domain-definition end-domain*
`domain` *factor* `(` *argument-seq*$_{opt}$ `)` *local-seq*$_{opt}$ *domain-definition end-domain*

local-seq:
`local` *local-var-seq* `;`
`local` *local-var-seq* `;` *local-seq*

domain-definition:
domain-declaration-seq$_{opt}$ *domain-entry-seq*$_{opt}$ *initializer*$_{opt}$

domain-declaration-seq:
domain-declaration `;`
domain-declaration `;` *domain-declaration-seq*

domain-declaration:
`inherits` *expression*
`category` *expression-seq*
`axiom` *expression-seq*

domain-entry-seq:
name `:=` *statement* `;`
name `:=` *statement* `;` *domain-entry-seq*

initializer:
`begin` *statement-seq*

end-domain:
`end`
`end_domain`

The first *factor* defines the name of the constructor. It must be valid as a left-hand-side of an assignment. Only one domain may be inherited from.

category-constructor:
`category` *factor local-seq*$_{opt}$ *category-definition end-category*
`category` *factor* `(` *argument-seq*$_{opt}$ `)` *local-seq*$_{opt}$ *category-definition end-category*

category-definition:
category-declaration-seq$_{opt}$ *category-entry-seq*$_{opt}$ *initializer*$_{opt}$

category-declaration-seq:
category-declaration `;`
category-declaration `;` *category-declaration-seq*

category-declaration:
`category` *expression-seq*
`axiom` *expression-seq*

category-entry-seq:
name `;`
name `:=` *statement* `;`
name `;` *entry-seq*
name `:=` *statement* `;` *entry-seq*

end-category:
`end`
`end_category`

axiom-constructor:
`axiom` *factor local-seq*$_{opt}$ *initializer*$_{opt}$ *end-axiom*
`axiom` *factor* `(` *argument-seq*$_{opt}$ `)` *local-seq*$_{opt}$ *initializer*$_{opt}$ *end-axiom*

end-axiom:
`end`
`end_axiom`

5.19 Parallel Statements

Currently, neither SciFace Software nor the MuPAD Group at the University of Paderborn have the resources to further develop the concepts of micro-parallelism for MuPAD. These were intended to allow for an easy parallel programming on shared-memory architectures supported by parallel language constructs. Thus, the parallel language constructs and their underlying system functions have been removed from the kernel. These constructs were the parallel for-loops and the parallel statement sequence. As a consequence, also the sequential statement sequence (`seqbegin` ... `end_seq`) has been removed.

The concepts of macro-parallelism (parallel execution of loosely coupled kernels communicating via messages) are currently re-designed by the MuPAD Group. Because this is a research project, the former specifications of macro-parallelism have been withdrawn. The functions intended to implement macro-parallelism (`global`, `topology` and the functions for pipes and queues) have been removed from the production kernel.

5.20 Debugging

The trace functionality intended to be used for debugging (command line option `-t` and function `trace`) have become obsolete because of the debugger. They have been removed. The same holds for the special identifiers `ERRORLEVEL` and `PRINTLEVEL`. (Please note that there exists a tracing function `prog::trace` which produces more compact output than the former kernel function `trace`.)

Overview

A summary of the language changes for version 2.0 as described in the previous sections:

New Features

- lexical scoping for procedures, new option `escape`,
- programmatic variables (`DOM_VAR`) vs. identifiers (`DOM_IDENT`),
- saving of identifiers via `save`,
- default values for parameters of procedures,
- type declarations for local variables,
- access to the domain a procedure belongs to via `dom`,
- functions may be defined via the arrow-operator `->`,
- new operator `in`,
- user-defined operators,
- slots may hold values of objects,
- arbitrary strings as names,
- special syntax for domain-, category- and axiom constructors,
- control statements and procedure definitions may end with `end`,
- conditional compilation with `%if`

Changes

- `args` returns the current value of the variable holding an input parameter,
- the index used with the `$`-operator must be a name, not an expression,
- `this` is replaced by the variable `dom` in the domains package,
- `procname` is an implicitly declared variable,
- `domattr` and `funcattr` have been replaced by `slot`,
- use `delete` instead of `unassign` to delete values,
- `eval` works for any argument,
- `subs` and `subsex` do no longer change domains,
- `history` gives access to previous input and output,

- input needs not end with `;` or `:`,
- statements need not be enclosed in brackets,
- minor grammar changes.

Removed Features

- no pure functions any longer (the functions `fun`, `func`, `block`, `newpurefunc` and `purefuncarg` were removed),
- `PRINTLEVEL`, `ERRORLEVEL`, `EVAL_STMT` were removed,
- the `HISTORY` for procedures may no longer be changed,
- keywords are no longer allowed as names for slots entered via the `::-` operator,
- `&` and `bin_op` have been removed in favor of `operator`,
- parallel statements and functions were removed,
- tracing via `trace` or the `-t` command line option were removed.