Project calculator

Helge Janicke

July 3, 2002

Guide to Calculator

This is the documentation for Calculator for those who want to use. The first chapters cover the features and defined operations.
For questions, bugs or ideas mail to: hj@janicke.de

or visit: www.furchur.de

Contents

1 Overview **4**

2 Newbee to Calculator? **4**

3 Supported operations **4**
3.1 Simple arithmetic operations . . . 4
3.1.1 factorial . . . 4
3.1.2 unary plus, minus . . . 5
3.1.3 multiplication . . . 5
3.1.4 division . . . 6
3.1.5 mod and remainder . . . 6
3.1.6 sum and difference . . . 6
3.2 Simple trigonometric operations . . . 7
3.2.1 (a)sin, (a)cos, (a)tan . . . 7
3.2.2 atan2 . . . 7
3.2.3 toDegrees, toRadians . . . 7
3.3 java.lang.Math miscellaneous operations . . . 8
3.3.1 random . . . 8
3.3.2 rint, round . . . 8
3.3.3 sqrt . . . 8
3.3.4 pow . . . 9
3.3.5 log . . . 9
3.3.6 exp . . . 9
3.3.7 min, max . . . 9
3.3.8 floor, ceil . . . 9
3.4 Comparision and relational operations . . . 10
3.4.1 comparision in detail . . . 10
3.5 Lists and sets . . . 10
3.5.1 Creating lists . . . 11
3.5.2 LIST and SET operator . . . 11
3.5.3 append . . . 12
3.5.4 remove . . . 12
3.5.5 addAll . . . 12
3.5.6 retainAll . . . 13
3.5.7 containsAll . . . 13
3.5.8 sizeof . . . 13
3.5.9 $ . . . 13
3.5.10 set . . . 13
3.5.11 first, last . . . 13
3.5.12 sort . . . 14
3.5.13 binsearch . . . 14
3.6 Logical operations . . . 14
3.7 Matrix operations . . . 14
3.7.1 dot . . . 15
3.7.2 gj . . . 15
3.7.3 lu . . . 15
3.7.4 invert . . . 16
3.7.5 lp . . . 16
3.8 Type cast operations . . . 17
3.9 Miscellaneous operators . . . 17
3.9.1 gcd . . . 17
3.10 Lowlevel operators . . . 17
3.10.1 if then else . . . 18
3.10.2 typeof . . . 18

4 Variables **19**
4.1 Creating variable entries . . . 19
4.2 Using namespace layers . . . 19
4.2.1 Creating new namespace layers . . . 19
4.2.2 Dropping namespace layers . . . 20
4.2.3 Example for using namespace layers . . . 20

5 Reusing formulas **22**
5.1 Using the Input operator . . . 22
5.1.1 Example: The input operator . . . 22
5.2 Using functions . . . 22
5.2.1 Creating functions . . . 22
5.2.2 Passing parameters to functions . . . 22
5.2.3 Scope of variables in functions . . . 23
5.2.4 Recursion . . . 23
5.2.5 How the evaluation is done . . . 23
5.3 Example: Load fibunacci function from file . . . 23

List of Figures

1 Example: Using namespace layers . . . 21
2 fibonacci.txt . . . 24

List of Tables

1 Arithmetic operations . . . 4
2 Trigonometric operations . . . 7
3 java.lang.Math miscellaneous operations . . . 8
4 Comparision and relational operations . . . 10
5 List and set operations . . . 11
6 Logical operations . . . 14
7 Matrix operations . . . 15
8 Type cast operations . . . 17
9 Miscellaneous operations . . . 17
10 Lowlevel operations . . . 18

1 Overview

This documentation covers both, a userguide to Calculator and an programmers guide, for those who want to enhance Calculators functionality or support new Operators.
For those who want to understand how Calculator can be used, I would suggest to read the first 5 sections. That should give you an overview of what you can do with Calculator and what you cannot.

For those who plan to invent new operators section **??** and **??** is a must. The first covers the syntax of the configuration file that is used to set up operators precedence and association. The second gives a short step by step introduction for creating new operators, acompanied by two small examples.
The other sections cope with Calculators inner structure and show the underlying concepts. There are some examples that help you understand when an Operator is invoked and what happend before. You may find it worth reading, too.

2 Newbee to Calculator?

Calculator is a JAVA program to interpret formulas. It first was designed to evaluate simple formulas as `2 + 3 - 5 * 4`, but evolved rapidly. The main goal (beside correct evaluation) was to make it easy to define new operators and increase the functionality without much effort.
But now Calculator comes along with a plenty of operations, type correct evaluation, and support for matrices, variable and (recursive) function definitions. This is far more than it was originally designed for.

The next sections will give you an overview of the capabilities and supply plenty of examples.

3 Supported operations

The available operations are defined by an configuration file supplied with the Calculator package. This section only describes the operators that are defined in the configuration file printed in this document (see Appendix **??**).

3.1 Simple arithmetic operations

Functor	precedence	association	usage	description
`!`	30000	LEFT	POSTFIX	factorial (see 3.1.1)
`+`	10000	RIGHT	PREFIX	unary plus (see 3.1.2)
`-`	10000	RIGHT	PREFIX	unary minus (see 3.1.2)
`*`	1000	LEFT	INFIX	multiplication (see 3.1.3)
`/`	1000	LEFT	INFIX	division (see 3.1.4)
`mod`	1000	LEFT	INFIX	modulo (see 3.1.5)
`remainder`	1000	LEFT	INFIX	IEEE remainder (see 3.1.5)
`+`	500	LEFT	INFIX	sum (see 3.1.6)
`-`	500	LEFT	INFIX	difference (see 3.1.6)

Table 1: Arithmetic operations

3.1.1 factorial

The factorial operator does define evaluation for **int** and **double** values. The result is always of type **double**. Mind that when using with a double (x) the following equation decides wheather the factorial is checked:

$$x == int\,(x)$$

If this check fails the following error is reported:

factorial not defined for floating point values

You may pass factorial a matrix type. The operation is performed elementwise on the matrix, where each element must satisfy the equation above. The result is a new matrix with an identical structure, containing double value.

EXAMPLE

```
1) input :   4!                  factorial for an  int value
   result:  24.0                 result is an  double value
2) input :   3!!                 factorial is left associative 3!! = (3!)! = 6.0!
   result: 720.0
3) input :   3.0!                factorial for an  double value
   result:   6.0
4) input : {2, 3, 4}!            elementwise factorial for a vector
   result: {2.0, 6.0, 24.0}      result is a vector
```

3.1.2 unary plus, minus

The unary plus and unary minus operators normally are used for the sign of a value. They both are right associative, which means you can cascade them. The return type is always the same as the arguments' type, for all scalar types that are currently supported by Calculator. You may use both operators with matrix types. The operation is then performed elementwise (which is the common sense). The result will be a matrix having the same type and structure as the argument.

EXAMPLE

```
1) input :   -4                  negation
   result:   -4
2) input :  --4                  cascaded use
   result:    4
3) input : -{2, 3, 4}            elementwise negation for a vector
   result:  {-2, -3, -4}         result is a vector
```

3.1.3 multiplication

scalar x *** scalar** y The multiplication operator can be used for all scalar types currently supported by Calculator. When passing two arguments of the same type, this type is returned. When passing different types an automatic widening conversion is performed.

wider type `double, float, long, int` *narrower type*

EXAMPLE

```
1) input :   3*4                 int * int
   result:   12                  int
2) input :   3.0f * 3            float * int
   result:   9.0f                float
```

scalar x *** matrix** A**, matrix** A *** scalar** x Multiplication of a scalar with a matrix is performed elementwise. Each element of the matrix is multiplicated with x.
Currently this multplication is only defined for the scalar type **double**. This is why the result is a matrix containg double values. If you still need int values, convert the double matrix to int matrix using: `int ({3.0, 6.0, 9.0})` (see 3.8)

EXAMPLE

```
1) input :   3*{1,2,3}          int * matrix
   result:   {3.0, 6.0, 9.0}    matrix
2) input :   {1,2,3}*3          matrix * int
   result:   {3.0, 6.0, 9.0}    matrix
```

matrix A *** matrix** B *This operation is deprecated, it will be replaced by elementwise multiplication of listelements*
THIS IS NO MATRIX MULTIPLICATION!
Both A and B must have the same structure. Each element $A_{ijk...}$ is multiplicated by $B_{ijk...}$. The result is a matrix having the same structure as A and B. The multiplication of the matrix elements follows the rules described in paragraph 3.1.3.

EXAMPLE

```
1) input :   {3,2,1}*{1,2,3}    matrix * matrix
   result:   {3, 4, 3}          matrix
```

3.1.4 division

Division is defined for all scalar types. The result of the division is the same as the argument type. If the arguments are from different type a widening conversion is made (see multiplication 3.1.3). Be careful when dividing int values, the result is an int value, too.
You may divide a matrix type by an scalar. All elements are divided by the scalar. Note that the division is only defined for double values. All others dividents are extended to a equivalent double value before the division is performed.

EXAMPLE

```
1) input :   3.0/4.0            double / double
   result:   0.75               double
2) input :   3/4                int / int
   result:   0                  int, truncated
3) input :   {1,2,3,4}/2        matrix / int
   result:   {0.5,1.0,1.5,2.0}  matrix withvalues
```

3.1.5 mod and remainder

`mod` returns the remainder of an integer division.
`remainder` returns the remainder of an division described in IEEE 754. Dividend and divisor are both double values.

3.1.6 sum and difference

scalar $x \pm$ **scalar** y These operations are defined for all scalar types supported by Calculator. If x and y are from different type a widening conversion is made (see multiplication 3.1.3).

EXAMPLE

```
1) input :   3.0+4.0            double + double
   result:   7.0                double
2) input :   3-4.0              int - double
   result:   -1.0               double, after widening conversion
```

matrix $x \pm$ **matrix** y The operations are also defined for matrix types. Both matrices must have the same structure. The operation is performed elementwise, and widening conversions are made if neccassary. The result is a matrix having the same structure as the arguments. The type of the matrix is the widest component type.

EXAMPLE

```
1) input :   {3, 4} - {4, 3}      matrix - matrix
   result:   {-1, 1}              matrix
2) input :   {3, 4.0} - {4, 3}    matrix - matrix
   result:   {-1, 1.0}            matrix
```

3.2 Simple trigonometric operations

Functor	precedence	association	usage	description
sin	10000	RIGHT	PREFIX	sin
cos	10000	RIGHT	PREFIX	cos
tan	10000	RIGHT	PREFIX	tan
asin	10000	RIGHT	PREFIX	asin
acos	10000	RIGHT	PREFIX	acos
atan	10000	RIGHT	PREFIX	atan
atan2	10000	RIGHT	PREFIX	atan2
toDegrees	10000	RIGHT	PREFIX	radians → degrees
toRadians	10000	RIGHT	PREFIX	degrees → radians

Table 2: Trigonometric operations

3.2.1 (a)sin, (a)cos, (a)tan

These are all prefix operations for one argument, which is a double representing an angle in radians. If you are passing other scalar types than double, a widening conversion is performed. The result is a double value.

EXAMPLE

```
1) input :   sin(0.0)
   result:   0.0
```

Return values for:

asin are in $[-\frac{\pi}{2} ; \frac{\pi}{2}]$

acos are in $[0 ; \pi]$

atan are in $[-\frac{\pi}{2} ; \frac{\pi}{2}]$

3.2.2 atan2

`atan2(x,y)` converts rectangular coordiantes (x,y) to polar (r,θ). This operator returns the θ value, which is in $[-\pi ; \pi]$. The actual behaviour is determined by the java.lang.Math implementation of `atan2`.

3.2.3 toDegrees, toRadians

These two operations convert between radians and degree measuring.

3.3 java.lang.Math miscellaneous operations

Functor	precedence	association	usage	description
`random`	10000	RIGHT	CONSTANT	random double in [0 ; 1]
`rint`	10000	RIGHT	PREFIX	double closest to mathematical integer
`round`	10000	RIGHT	PREFIX	rounds to double → long or float → int
`sqrt`	10000	RIGHT	PREFIX	the squareroot of argument
`pow`	10000	RIGHT	PREFIX	the power x^{exp}
`log`	10000	RIGHT	PREFIX	natural logarithm
`exp`	10000	RIGHT	PREFIX	exponential funtion
`min`	10000	RIGHT	PREFIX	minimum of two values, or of a List
`max`	10000	RIGHT	PREFIX	maximum of two values, or of a List
`floor`	10000	RIGHT	PREFIX	largest mathematical integer not greater than argument.
`ceil`	10000	RIGHT	PREFIX	lowest mathematical integer not less than argument.

Table 3: java.lang.Math miscellaneous operations

3.3.1 random

Returns an random double value which in [0 ; 1]

EXAMPLE

```
1) input:  int(random * 100)      returns a random int in [0;100]
   result: 92
```

3.3.2 rint, round

Rounds the result to the closest int value. `rint` returns the double closest to a mathematical integer.

EXAMPLE

```
1) input:  round(3.1415927)
   result: 3                      result is an int
2) input:  rint(3.1415927)
   result: 3.0                    result is a double
```

3.3.3 sqrt

Computes the square root for the argument. The argument is expected to be an double, but if it isn't a widening conversion is made. The result is always a double.

EXAMPLE

```
1) input:  sqrt(2)
   result: 1.4142135623730951
```

3.3.4 pow

Computes x^{exp} both x and y are expected to be double values. But if neccassary a widening conversion is made. The result is always a double.

EXAMPLE

```
1) input:  pow(2,4)
   result: 1024.0
```

3.3.5 log

Computes the natural logarithm of the argument. If the argument is zero `-Infinity` is returned, for negative values the operation returns `NaN`.

EXAMPLE

```
1) input:  log(2)
   result: 0.6931471805599453
2) input:  log(0)
   result: -Infinity
```

3.3.6 exp

Returns Euler's number e raised by the argument, which is expected to be a double, but widened otherwise.

EXAMPLE

```
1) input:  exp(2)
   result: 7.38905609893065
```

3.3.7 min, max

Returns the minimum/maximum of two values, or of a list. If the arguments of different type, a widening conversion is made.

EXAMPLE

```
1) input:  max(2.0, 4)
   result: 4.0
2) input:  min [10, 3, 4, 5, 6, -7, 8, 9]
   result: -7
```

3.3.8 floor, ceil

Returns the closest to a mathematical integer double value, not larger (*floor*) or not less (*ceil*) than the argument.

EXAMPLE

```
1) input:  floor(2.5)
   result: 2.0
2) input:  ceil(2.5)
   result: 3.0
```

3.4 Comparision and relational operations

Functor	precedence	association	usage	description
==	400	LEFT	INFIX	true if both values are equal
<>	400	LEFT	INFIX	true if both values are not equal
>=	410	LEFT	INFIX	true if leftvalue is greater tahn or equal rightvalue
<=	410	LEFT	INFIX	true if leftvalue is less than or equal rightvalue
>	410	LEFT	INFIX	true if leftvalue is greater than rightvalue
<	410	LEFT	INFIX	true if leftvalue is less than rightvalue

Table 4: Comparision and relational operations

3.4.1 comparision in detail

Both arguments are expected to have the same type, if not they are widened.
The result of a comparasion is always a truth value.
It is possible to apply the comparision operator to matrices. Therefore both matrices need to have equal dimensions. The matrices are compared elementwise and the result will be a truth matrix having the same dimensions as the compared matrices. The componenttype of the result will be boolean.
It is possible to compare a truth matrix with a single boolean value x. The result of such a comparision is `true` if and only if all elements compared to x will result in `true` .

Comparision for lists is currently not supported.

EXAMPLE

```
1) input:  2 > 1
   result: true
2) input:  2 >= 2
   result: true
3) input:  {1, 2, 3, 4} == {1, 3, 2, 4}
   result: {true, false, false, true}
4) input:  {1, 2, 3, 4} == {1, 3, 2, 4} == false
   result: false
```

3.5 Lists and sets

You generate lists and sets and apply primitive operations to both. Elementwise usage of the previous described operators is not yet implemented. The next section describes how to generate lists and sets.

Functor	precedence	association	usage	description
SET	10000	RIGHT	PREFIX	Converts a list into a set.
LIST	10000	RIGHT	PREFIX	Converts a set int a list.
append	10000	LEFT	PREFIX	appends a value to an list.
remove	10000	LEFT	PREFIX	removes the value at index from a list
addAll	10000	LEFT	INFIX	adds a list/set to another list/set
retainAll	10000	LEFT	INFIX	retains all elements that are not in passed list/set
containsAll	10000	LEFT	INFIX	true if all elements of the passed list/set are in the list
contains	10000	LEFT	INFIX	true if the element is in list
sizeof	10000	LEFT	PREFIX	returns the size of the list/set
first	10000	LEFT	PREFIX	returns the first element of the list
last	10000	LEFT	PREFIX	returns the last element of the list
set	10000	LEFT	PREFIX	sets the value at index in the list to a new value
$	5000	LEFT	INFIX	returns the element at index
sort	10000	LEFT	PREFIX	sorts the list
binsearch	10000	LEFT	PREFIX	returns the index of the element in list
sublist	10000	LEFT	PREFIX	returns a sublist

Table 5: List and set operations

3.5.1 Creating lists

To generate a list use the square brackets `[]`. The elements of the list should be separated with commatas. Unlike matrices (vectors) a list can contain any data. Here is an example:

EXAMPLE

```
1) input:  []            an empty list
   result: []
2) input:  [1,2,3,4]
   result: [1, 2, 3, 4]
```

If you want to convert a list to an vector (matrix) simply surround the list with curly braces. That means writing: `{1,2,3,4}` is equal to `{[1,2,3,4]}`.

3.5.2 LIST and SET operator

You can transform a list into a set, and every set into a list. These very specialised typecasts are done with the `LIST` and `SET` operators. Mind that these commands are uppercase.

EXAMPLE

```
1) input:  a = SET [1,2,3,4,4,4,5]     a set contains each element only once
   result: a -> [5, 4, 3, 2, 1]
2) input:  b = LIST a
   result: b -> [5, 4, 3, 2, 1]
3) input:  a addAll [1,2,3,4]
   result: false                       no elements were added
4) input:  b addAll [1,2,3,4]
   result: true                        elements are added
5) input:  a
   result: a -> [1, 2, 3, 4, 5]
6) input:  b
   result: b -> [5, 4, 3, 2, 1, 1, 2, 3, 4]
```

3.5.3 append

To append an element to the list use `append`.

EXAMPLE

```
1) input:  a = [1,2,3,4,5]
   result: a -> [1, 2, 3, 4, 5]
2) input:  append(a,6)
   result: [1, 2, 3, 4, 5, 6]
```

3.5.4 remove

Removes an element from the list.

EXAMPLE

```
1) input:  a = [11,12,13,14,15]
   result: a -> [11, 12, 13, 14, 15]
2) input:  remove(a,3)                  remove 4th element
   result: [11, 12, 13, 15]
```

3.5.5 addAll

`a addAll b` This operator adds a list/set `b` to another list/set `a`.

EXAMPLE

```
1) input:  a = [1,2,3]
   result: a -> [1, 2, 3]
2) input:  b = SET [2,3,4,2]
   result: b -> [4, 3, 2]
3) input:  c = a
   result: c -> [1, 2, 3]
4) input:  a addAll b
   result: true
5) input:  c
   result: c -> [1, 2, 3, 4, 3, 2]
```

3.5.6 retainAll

`a retainAll b`. This operator removes all elements from a, that are not in b. Both a and b must be either sets or lists.

EXAMPLE

```
1) input:  [1,2,3,4,5] retainAll [4,5,6,7,8]
   result: [4, 5]
```

3.5.7 containsAll

`a containsAll b`. Returns true if all elements from b are also elements of a. Both a and b must be either sets or lists.

EXAMPLE

```
1) input:  [1,2,3,4,5] containsAll [4,5]
   result: true
2) input:  [1,2,3,4,5] containsAll [0,4,5]
   result: false
```

3.5.8 sizeof

`size a` returns the current size of list/set `a`.

EXAMPLE

```
1) input:  sizeof [1,2,3,4,5]
   result: 5
```

3.5.9 $

`a$n` returns the n_{th} element of list `a`. n is zero based.

EXAMPLE

```
1) input:  [-1,-2,-3,-4,-5]$3
   result: -4
```

3.5.10 set

`set a, i, b` sets the i_{th} element of list `a` to `b`.

EXAMPLE

```
1) input:  a = [1, 2, 3, 4, 3, 2]
   result: a -> [1, 2, 3, 4, 3, 2]
            replaces first occurence of 3 with 13
2) input:  set (a, binsearch(a, 3), 13)
   result: [1, 2, 13, 4, 3, 2]
```

3.5.11 first, last

`first a` is equal to `a$0`

`last a` is equal to `a$(sizeof(a)-1)`

3.5.12 sort

`sort a` sorts the list a using mergesort.

3.5.13 binsearch

`binsearch(a,b)` returns the first index of `b` in list `a`. It returns a negative value if `a` doesn't contain `b`.

3.6 Logical operations

Note that all operators of this group can be applied to a matrix type and then perform elementwise evaluation.

Functor	precedence	association	usage	description
`true`	900000	-	CONSTANT	truth value for true
`false`	900000	-	CONSTANT	truth value for false
`AND`	240	LEFT	INFIX	true if both arguments are true
`OR`	230	LEFT	INFIX	true if at least one argument is true
`XOR`	240	LEFT	INFIX	true if one argument is true
`NOT`	10000	RIGHT	PREFIX	true if argument is false
`COMPL`	10000	RIGHT	PREFIX	integers bitwise complement
`SHR`	490	LEFT	INFIX	shift right
`SHL`	490	LEFT	INFIX	shift left
`USHR`	490	LEFT	INFIX	unsiged shift right

Table 6: Logical operations

3.7 Matrix operations

First a short explanation of matrices, and what is meant by matrix type. A matrix type can be an vector, a matrix (order = 2), or a cube (order = 3) or some matrix (order = n). A matrix always have a component typ. This is a typ that is applicable for all components of the matrix.
It is possible to generate different matrices for a different content type and order. Which matrix to create when is defined in the configuration file. For example having a list of ints [1,2,3,4] the default configuration creates an IntegerMatrix1D from it.
You can create matrices using the curly braces:

EXAMPLE

```
1) {1,2,3,4,5}              generates IntegerMatrix1D
2) {1.0, 2.0, 3.0, 4.0}     generates DoubleMatrix1D
3) { {1, 2}, {2, 1}}        generates IntegerMatrix2D
4) {{{1.0}}}                genetates DoubleMatrix (order = 3)
```

Currently there is very little support to retrieve values from matrices, explicitly change them, or create subsets of matrices. Maybe in future versions there will be operators doing this job. But some algorithms are already implemented.

Functor	precedence	association	usage	description
`dot`	1000	LEFT	INFIX	Matrix (Tensor) produkt
`gj`	10000	RIGHT	PREFIX	Gauss Jordan algorithm
`lu`	10000	RIGHT	PREFIX	LU decomposition
`invert`	10000	RIGHT	PREFIX	inverts a matrix
`lp`	10000	RIGHT	PREFIX	solves a linear program

Table 7: Matrix operations

3.7.1 dot

`A dot B` is the matrix produkt of A and B.
A and B must satisfy the following equations

$$C_{ij...l...m} = A_{ij...k} \; dot \; B_{kl...m}$$

This means the last dimension of A must equal the first dimension of B. The result matrix has an order of

$$oder(A) + order(B) - 2$$

This means the matrix produkt of two vectors is a scalar value. The matrix type is determined as explained above.

3.7.2 gj

The Gauss Jordan algorithm needs two parameters. The first parameter is the NxN matrix containing the coefficients. The second parameter is an NxM matrix containing M column vectors with right hand side values.
The result is a list containing two elements. First is a scrambled inverted matrix of the first parameter, the second is a matrix containing the solution for this equation set.

NOTE: gj expects DoubleMatrix2D to be passed as arguments.

gj does full pivoting, but the solution is **not** improved iteratively.

EXAMPLE

```
1) input:  C = { {1.0, 2.0, 3.0}, {2.0, 1.0, 1.0}, {0.0, 0.0, 2.0} }
2) input:  R = { {2.0}, {3.0}, {1.0} }
3) input:  result = (gj(C,R))$1      interrested in solution
   result: result -> {{1.5}, {-0.5}, {0.5}}
4) input:  C dot result              check correctness
   result: {{2.0}, {3.0}, {1.0}}
```

3.7.3 lu

As the Gauss Jordan algorithm the LU-Decomposition expects two parameters. The first is the NxN matrix containing the coefficients, the second is either a (N) row vector, containing the righthandside values, or a list of row vectors.
The advantage of the list is that the actual LU-Decomposition is only made once for the coefficient-matrix.

NOTE: lu expects a DoubleMatrix2D, and a (list of) DoubleMatrix1D to be passed as parameters.

EXAMPLE

```
1) input:  C  = { {1.0, 2.0, 3.0}, {2.0, 1.0, 1.0}, {0.0, 0.0, 2.0} }
2) input:  R1 = { 2.0, 3.0, 1.0 }
3) input:  R2 = { 1.0, 3.0, 2.0 }
3) input:  result = lu(C,[R1, R2])
   result: result -> [{1.5, -0.5, 0.5}, {2.0, -2.0, 1.0}]
4) input:  C dot result$0              check correctness
   result: {2.0, 3.0, 1.0}
4) input:  C dot result$1
   result: {1.0, 3.0, 2.0}
```

3.7.4 invert

The invertation a a matrix is done using the LU-Decomposition algorithm. `invert` expects one argument, a NxN DoubleMatrix2D.

EXAMPLE

```
1) input:  C  = { {1.0, 2.0, 3.0}, {2.0, 1.0, 1.0}, {0.0, 0.0, 2.0} }
2) input:  result = invert C
   result: result ->
   {{-0.3333333333333333,  0.6666666666666666,  0.16666666666666669},
    { 0.6666666666666666, -0.3333333333333333, -0.8333333333333334},
    { 0.0,                 0.0,                 0.5}}
3) input:  C dot result                check correctness
   result: {{1.0, 0.0, 0.0}, {0.0, 1.0, 0.0}, {0.0, 0.0, 1.0}}
```

3.7.5 lp

Solves a linear program. This algorithm is used to find an optimum for a target function restricted by an equation set. The algorithm expects three parameters:

1. NxN Matrix containing the coefficients (DoubleMatrix2D).
2. N row vector containing the target functions coefficients (DoubleMatrix1D).
3. N row vector containing the righthandside values of the equation set.

The result is a String showing information on the optimum.
lp uses the multiphase/simplex algorithm to solve the linear program.

EXAMPLE

```
1) input:  A = -1, -1, 0,  -1, 1,  -1
2) input:  B = -3,-2,3
3) input:  C = -1,-2
4) input:  lp(double(A),double(C),double(B))
   result:
   found optimum: -5.0 for:
   X[4] = 4.0
   X[1] = 2.0
   X[0] = 1.0
   all other x are 0
```

3.8 Type cast operations

Note that when applying these operators to matrix type the cast is performed elementwise. The returned matrix typ is generated according to the matrix creation rules mentioned in 3.7.

Functor	precedence	association	usage	description
`int`	10000	RIGHT	PREFIX	converts argument to an int
`long`	10000	RIGHT	PREFIX	converts argument to an long
`float`	10000	RIGHT	PREFIX	converts argument to an float
`double`	10000	RIGHT	PREFIX	converts argument to an double

Table 8: Type cast operations

EXAMPLE

```
1) input:  double(3)
   result: 3.0
2) input:  double {1,2,3,4}        convert IntegerMatrix1D
   result: {1.0, 2.0, 3.0, 4.0}    to DoubleMatrix1D
```

3.9 Miscellaneous operators

Here are some basic operations, which also serve as examples in later chapters.

Functor	precedence	association	usage	description
`lsb`	90000	-	CONSTANT	returns the least significant bit of an double
`gcd`	10000	LEFT	PREFIX	computes the greates common divisor

Table 9: Miscellaneous operations

3.9.1 gcd

To compute the greatest common divisor the simple Euklidean Algorithm is used. The algorithm expects 2 nonnegative integer values as parameters.

This operator is described in detail in section **??**

EXAMPLE

```
1) input:  gcd(21, 9)
   result: 3
```

3.10 Lowlevel operators

There are some operators, that work on a very low level. Most of these are described in later chapters.

Functor	precedence	association	usage	description
`if`	10000	LEFT	PREFIX	conditional statement
`create`	10000	RIGHT	PREFIX	creates namspacelayers and functions (see 4.2.1, 5.2.1)
`drop`	10000	RIGHT	PREFIX	drops namspacelayers and functions (see 4.2.2)
`=`	200	LEFT	INFIX	assigns a value to a variable (see 4.1)
`,`	100	LEFT	INFIX	separates list entries
`input`	10000	LEFT	PREFIX	executes formulas from a file (see 5.1)
`currentNamespace`	10000	-	CONSTANT	returns the current namespace layer (see 4)
`@`	5000	LEFT	INFIX	references a variable in a specific namespace layer (see 4.1)
`typeof`	1000	LEFT	PREFIX	shows the type of an expression

Table 10: Lowlevel operations

3.10.1 if then else

In order to support recursive function calls it was neccassary to implement conditional statements. It is a bit difficult to handle, because you cannot write `if then else` as in a programming language. Maybe in later versions there will be such a possibility.
`if` is defined as a prefix operator. It expects three arguments of type String. The first argument is the condition and must evaluate to a truth value. The second and third are the true and false branch of the conditional statement.

EXAMPLE

```
1) input:  a = random
                condition  true-branch  false-branch
2) input:  if ("a < 0.5", "a = 1 - a", "a")
```

3.10.2 typeof

If you are not certain which type an expression is of, use the typeof operator.

EXAMPLE

```
1) input:  a = [1,2,3,4,[5,6,7]]
2) input:  typeof a
   result: java.util.Vector
3) input:  typeof a$4
   result: java.util.Vector
4) input:  typeof a$4
   result: de.janicke.hj.calc.matrix.IntegerMatrix1D
```

4 Variables

Calculator supports variables. That means you can store results for later usage. For example the following line assigns *10* to a variable named *a*.

```
a = 10
```

When talking of variables there are a few basic things you have to know.

1. Variables are kept in namespace layers. A variables' name is unique within a namespace layer.
2. A namespace consists of one or more namespace layers. All namespace layers are ordered using a stack. When refering to the *current layer* the topmost layer is meant.
3. Resolving a variable means to replace the variables' name with an object of type Var. The whole namespace is searched for a variable entry (searching all namespace layers from top).
4. Unwrapping a variable means to replace an object of type Var by the result of its' `getValue()` method.
5. Variables aren't type specific (means you can assign the vector `{1,2,3}` to *a*, also).
6. Variable names are case sensitive.

4.1 Creating variable entries

If you have never assigned a value to *a* before (in the current namespace layer) a new entry is created in the topmost namespace layer by the Assign operator `=`.
The second way of creating a new variable entry is the SpecificVar operator `@`. This operator is used as an infix operator, the left operand must be an unresolved word, the right one something thats' string representation equals a namespace layers' name. The result is of type Var. This operator tries to resolve a variable name exactly in the specified namespace, if there is no entry in this namespace a new entry is created.

4.2 Using namespace layers

Namespace layers are invented, to allow to write sets of formulars, that can used without affecting the values of already existing variables.
A good example for the use is the Function operator. When writing functions you normally do not know the circumstances the function will be used. More complex functions may need variables for computation, but when using them you usually do not want to affect already existing variables.
The solution is to create a new layer before executing the function, which is exactly what the Function operator does. All variables created by the function are stored in this new layer. It is possible to use any variable name, even if the name was used in the penultimate namespace layer, without affecting previously defined variables.

4.2.1 Creating new namespace layers

Create a new namespace layer using the Create operator:

```
create("namespace","myLayer01")
```

This creates a new layer named "myLayer01". All variable entries that are created are now stored in "myLayer01".
Note that the DefaultCalculator does create a namespace layer named "GLOBAL" when it initializes the namespace.

4.2.2 Dropping namespace layers

To drop a namespace layer you can use the Drop operator:

```
drop("namespace","myLayer01")
```

This will remove the layer "myLayer01" from the namespace, all variables stored in this layer will be lost.

4.2.3 Example for using namespace layers

Executing the following following code demonstrates the behaviour of namespace layers:

Step 1:

Create three variable entries (a,b,c):

```
a = 10.0
b = 20.0
c = 30.0
```

When you now use a,b and c their values will be 10.0, 20.0 and 30.0 (Test this by simply typing the variable name).

Step 2:

Create a new namespace layer *myLayer01* and two variable entries (a,b):

```
create("namespace","myLayer01");
a = 10.1
b = 20.1
```

When you use a,b and c their values will be 10.1, 20.1 and 30.0. This means that when you resolve a you will retrieve a variable stored in *myLayer01*. For c you will retrieve a variable stored in the *GLOBAL* layer, for it cannot be resolved in *myLayer01*.

Step 3:

Now create a new layer *myLayer02* and a new entry for a:

```
create("namespace","myLayer02");
a = 10.2
```

The result will be $a \rightarrow 10.2, b \rightarrow 20.1, c \rightarrow 30.0$.

Try to explicitly address variables by typing:

`a@GLOBAL`	will result in $a \rightarrow 10.0$
`a@myLayer01`	will result in $a \rightarrow 10.1$
`a@myLayer02`	will result in $a \rightarrow 10.2$
`a`	will result in $a \rightarrow 10.2$

Step 4:

Now drop the *myLayer01* by typing:

```
drop("namespace","myLayer01");
```

When you now use a,b and c their values will be 10.2 (which is *a@myLayer02*), 20.0 and 30.0 (both in *GLOBAL)*.
Here is a figure showing the Namespace during the 4 steps:

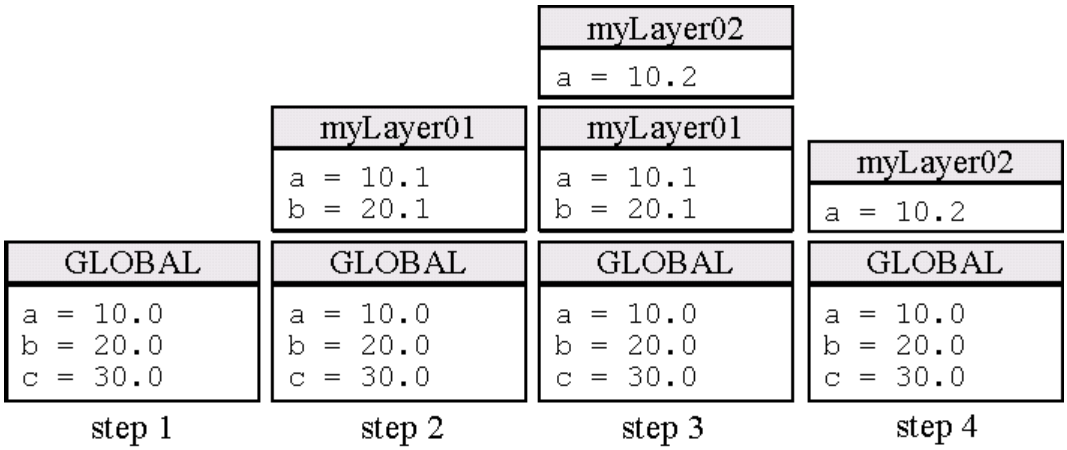

Figure 1: Example: Using namespace layers

5 Reusing formulas

There are two different ways you can reuse formulas you once wrote. The first is to store the formula in a file, and use the Input operator. The other way is to create a function that contains the formula.

5.1 Using the Input operator

The Input operator loads an ASCII input file and executes the formulas in this file.
Note that you can use C-style comments anywhere in you formula, except in Strings. If you want to store more than one formula or more complex operations in the file use the `;` command to seperate individual statements.
Formally the Input operator loads the file, creates a temporarily Function operator, and executes it without any parameters. After execution the result of the function is returned.

5.1.1 Example: The input operator

Having a file `foo.in` containing following statements:

```
a = 10;
b = 20;
a + b
```

A new Function operator is created for the three statements. The Function operator evaluates to 30, which is also the result of the Input operator (usage: `input("foo.in")`).

5.2 Using functions

Functions are used to spare time for often used computations (Spare time more in the sense of typing, the evaluation may take longer). A function is merely a String passed to the Function operator that represents a function.

5.2.1 Creating functions

As creating namespace layers, functions are created similiar:

```
create("function","test","a=10;b=20;a+b");
```

Where "test" is the name of the function, and "a=10;b=20;a+b" is the formula.
The create operator registers "test" as a prefix operator with a precedence = 10000. From now on you can use "test" as an operator.
When you create a new function named "test", also, the previous definition is overridden. You can explicitly remove a function definition by typing:

```
drop("function","test");
```

5.2.2 Passing parameters to functions

Some functions may need parameters to work properly. For it isn't a good style to work on previously defined variables directly you may want to pass parameters.
As functions are prefix operators the parameter list is passed as the operand. For a prefix operator needs one operand you have to pass at least empty paratheses `()` indicating an empty parameter list.
If you pass a parameter list (eg. `(10,20)`) an new variable entry `PARAM` is made in the functions namespace layer. For example you can rewrite the function "test" with:

```
create("function","test","PARAM$0+PARAM$1");
```

When you now use `test(10,20)` the result will be 30, too.

5.2.3 Scope of variables in functions

Everytime a function is evaluated a new namespace layer is created. Therefore variable usage doesn't affect already existant variables outside the function.
You may refer to other variables using the `@` operator, but be careful when doing so.
After the function has evaluated (or if it has failed) the namespacelayer of the function is droped.

5.2.4 Recursion

You may use recursive function calls, but be warned. It works slowly, as a lot of objects need to be created.
Ensure that your recursion has an definite end, or calculator will stop with an stack overflow.
The example below shows the use of recursion for the fibonacci function.

5.2.5 How the evaluation is done

Here's a shemata how the Function operator works:

1. store the current Evaluator module in *tmp*
   This is done to not affect the currently processed formula.
2. Create a new Evaluator module and register in Calculator.
3. Check if the function was processed before.
   If it was not:
   Convert the formula in a list of tokens (Resolver module).
   else:
   Reuse the previously created list of tokens.
4. Create a new namespace layer.
5. Evaluate the list of tokens.
6. Drop namespace layer.
7. Restore the Evaluator module with *tmp*.
8. Return the result.

Not to tokenize a functions formula twice is done for optimisation.

5.3 Example: Load fibunacci function from file

The following file contains the definition of the fibonacci function.

```
/** fibonacci numbers are defined by
 *  i)   f(0) = 0
 *  ii)  f(1) = 1
 *  iii) f(n) = f(n-1) + f(n-2) for n > 1 in N.
 *  fibonacci(n) returns a list of fibonacci numbers up to f(n)
 */
create ("function", "fibonacci",
        "my = currentNamespace;
         result@my = \"n must be >= 0 and in N\";
         n = PARAM$0;
         if(\"n == 0\", \"result@my = 0l\", \"\");
         if(\"n == 1\", \"result@my = 1l\", \"\");
         if(\"n >  1\", \"result@my = recFibonacci([0l,1l], (n-2))\", \"\");
         result@my
        ");


/** Recursion part of fibonacci. See fibonacci for more math details.
 *  recFibonacci(list, cnt) needs 2 parameters:
 *  param list:  a list containing at least f(0), f(1)
 *  param  cnt:  recursion counter
 */
create ("function","recFibonacci",

"if(\"PARAM$1 > 0\",    /*THEN*/
        // recursion, append new fibonacci and decrease cnt

        \"l=sizeof(PARAM$0)-1;
          recFibonacci(append(PARAM$0,((PARAM$0)$(l) + (PARAM$0)$(l-1))), PARAM$1 -1)
        \",

                        /*ELSE*/
        // recursion end, return list

        \"PARAM$0\"

                        /*ENDIF*/
)")

/* Note that the String parameters for the IF THEN ELSE are masked by an "\"
   because they appear within the formula string of an function definition.
*/
```

Figure 2: fibonacci.txt

Simply type: `input("fibonacci.txt")` to load the functions.
With `fibonacci(10)` you'll get the first 10 fibonacci numbers in a list.