ANALYTICS C# Development Library
Manual

Copyright © Sergey L. Gladkiy
Email: lrndlrnd@mail.ru
URL: www.Sergey-L-Gladkiy.narod.ru

Contents

Introduction

ANALYTICS is a C# library for developers. ANALYTICS library contains special classes to work with 'analytical' expressions in .NET programs – parse expressions, calculate expression values and so on.

ADVANTAGES of ANALYTICS library:
1. 100% C# code.
2. Strongly structured class hierarchy.
3. Universal algorithms (working with formulae of any complexity).
4. Many predefined functions.
5. Easy to introduce new functions for any argument types.
6. Easy to overload operators for any argument types.
7. Working with Complex numbers.
8. Working with Common Fractions.
9. Working with 3D vectors and tensors.
10. Working with physical values and units of measurement.
11. Working with indexed data (arrays, matrixes and higher dimensioned data).
12. Analytical derivative calculation.

Dependences

ANALYTICS C# library depends on:
1. NRTE (NET Run-Time Environment) library.

Extensions

There are the following extensions of ANALYTICS C# library:
1. ANALYTICS Complex.
2. ANALYTICS Fractions (depends on MATHEMATICS C# library).
3. ANALYTICS Linear Algebra (depends on MATHEMATICS C# library).
4. ANALYTICS Special (depends on MATHEMATICS C# library).
5. ANALYTICS Mathphysics (depends on MATHEMATICS and PHYSICS C# libraries).

1. Basic concepts

The main goal of ANALYTICS library is to provide easiest way to evaluate mathematical expressions. This part explains main concepts, used in ANALYTICS library to work with mathematics expressions and formulae.

Expressions

Mathematical expression is a sequence of elements for which its result value can be calculated. A simple example of expression is 'x+y' – knowing the values of x and y we can calculate their sum. Generally, expression contains such elements as **constants** (**literals**), **variables, operators, functions**. The result of an expression calculation depends on elements the expression consists of. For an example if 'x' and 'y' are real numbers (variables) – the result of the given sum expression is a real number, if one of the values is a complex value – result is a complex value too.

Literals

Literal is an unnamed constant value or a symbol staying for 'standard' constant value. For an example, in the expression **'2*(x+y)'** – **'2'** is a numerical literal. **Real** and C**omplex** literals are supported.

Real literals can be written in simple and exponential form. Simple form examples: **'100'**, **'1.23'**, **'-45.67'**[1]. Exponent form examples: **'1.23E-3'**, **'-2.45e+5'**.

For complex literals symbol **'I'** is used to denote **imaginary unit**. Complex literal consists of Real part and Imaginary part separated by **'+'** or **'-'**symbols. Real part is just a real literal, imaginary part is real literal plus imaginary unit symbol (**NOTE:** there is no multiplication symbol between real literal and imaginary unit symbol). Complex literal examples: **'I'**, **'-4I'**, **'2+3.2I'**, **'-2e2-4.45I'**, **'2.45+I'**, **'-1.2e-3+4.5e+2I'**.

The following 'standard' constants are recognized by parser: 'e' – Euler number, 'π', 'Pi' – Pi number, '∞' – positive infinity.

Variables

Variable is a named value (that can be changed during calculation). A variable name can include alpha-numeric symbols only and underscore symbol (the first symbol can be letter only). The value of a variable can be of any type. The type of a variable cannot be changed during calculation. The variable value is accessed via variable name. That is in an expression the variable name stands for the current variable value. For an example, let the variable **'A'** is a real variable whose current value is **'1.0'**, then the result of 'A+1' expression is **'2.0'**. Changing variable values allows calculation of the same expression for various values.

Operators

Syntactically **operator** is a symbol representing some mathematical operation. For an example, operator **'+'** stands for the sum operation. From the functional point of view, an operator implements some action with data values, called **operands**, and returns the result value.

All operators can be divided into many categories. The most common used are **unary** and **binary** operators. **Unary operators** have one operand and can be **prefix** (stands before its operand) and **postfix** (stands after its operand). An example of unary prefix operator is the **negation operator** ('-x', '-' is the operator, **'x'** is the operand). An example of unary postfix operator is the **factorial operator** ('n!', '!' is the operator, 'n' is the operand). **Binary operators** have two operands and commonly stand between them. An example of binary operator is the **addition operator** ('x+y', '+' is the operator, **'x'** and **'y'** are the operands).

Each operator has such attributes as **precedence** and **associativity**. The **precedence** determines the order of expression calculation. The operators with higher precedence applied to their operands before the operators with lower precedence. For an example, in the expression **'x+y*z'** the first operation performed is the multiplication of **'y'** and **'z'** and then the sum of **'x'** and the product result is calculated, because the **'*'** operator is of higher precedence than the **'+'** is. The order of calculation can be changed by using parantheses **'()'**. The **associativity** determines how operators with the same precedence are grouped in expressions. Let us consider the expression **'2^3^4'** (where the **'^'** operator stands for the power). The result of the expression depends on how the expression is interpreted – **'(2^3)^4=8^4'** or **'2^(3^4)=2^81'**. **Left** associativity means that operators are grouped from left to right (the first case), **right** associativity means grouping from right to left (the second case).

The following operators are defined in ANALYTICS library:
'+' addition operator (binary);
'-' subtraction operator (binary);
'*' multiplication operator (binary);
'/' division operator (binary);
'^' power operator (binary);
'•' dot operator (binary);
'-' negation operator (unary, prefix);
'~' tilde operator (unary, prefix);
'!' factorial operator (unary, postfix);
''' apostrophe operator (unary, postfix);
'≡' identically equal operator (binary);

---

[1] Current Culture Decimal separator is used in real literals.

'≈' approximately equal operator (binary);
'≠' not equal operator (binary);
'>' greater than operator (binary);
'<' less than operator (binary);
'≥' greater than or equal operator (binary);
'≤' less than or equal operator (binary);
'&' logical and operator (binary);
'\' logical or operator (binary);
'¬' logical not operator (unary, prefix);
'?' question operator (unary, prefix);
'#' number operator (unary, prefix);
Also the following 'Special'[2] operators defined:
'←' left arrow operator (binary);
'→' right arrow operator (binary);
'↑' up arrow operator (binary);
'↓' down arrow operator (binary);
'↔' left-right arrow operator (binary);
'↕' up-down arrow operator (binary);
'∂' derivative operator (unary, prefix);
'∫' integral operator (unary, prefix);
'Δ' delta operator (unary, prefix);
'∑' sum operator (unary, prefix);
'∏' product operator (unary, prefix);

General rules for all operators:
- Binary algebraic operators (+, -, *, etc.) have precedence as determined with standard mathematical rules.
- Relational operators (binary) have lower precedence than algebraic ones.
- Binary Logical operators have lower precedence than relational ones.
- Arrow operators (binary) have higher precedence than power operator.
- Binary operators are all left-associative.
- Unary operators have higher precedence than binary operators.
- Postfix operators have higher precedence than prefix operators.
The standard mathematical operators have default implementation for real operand types. Operators for other operands (complex, 3D vectors and tensors and so on) implemented in ANALYTICS extension libraries (see above).
All predefined operators can be overloaded (defined) for any other operand types. The following restrictions are applied for operator overloading:
- new operators cannot be defined;
- number of operands and attributes of operators (precedence and associativity) cannot be changed.

Functions

Syntactically **function** can be considered as a named operation with some data values – arguments. An example is the sine function '**sin(x)**', where 'sin' is the name of the function and 'x' is the argument. The function name determines what operation is performed with the arguments.
In addition to arguments, a function can have parameters. For an example, the logarithm of '**a**' to base '**b**' function **log$_b$(a)** has one argument '**a**' and one parameter '**b**'. Semantically parameters have the same meaning as the arguments have – data values on which the operation is performed. Syntactically (in ANALYTICS library) parameters are enclosed in braces '**{}**'.
General rules for functions:
- function arguments are always enclosed in parantheses '**()**';
- function parameters are always enclosed in braces '**{}**';
- if the function has no parameter, parameter braces are not obligatory, argument parantheses are always obligatory;
- function can have many parameters and/or many arguments;
- function arguments and parameters are separated by spaces ' ';
- there can be many functions with the same name and different number and/or type of arguments and/or parameters.
Some examples of syntactically correct function expressions:
max(a b) – maximum function of two arguments;
random() – random function with no arguments;
log{b}(a) – logarithm of '**a**' to base '**b**' (one parameter '**b**', one argument '**a**');
P{n m}(x) - the Associated Legendre function (two parameters '**n**' and '**m**' and one argument '**x**').
The ANALYTICS library contains many predefined functions of real arguments. Functions for other argument types implemented in ANALYTICS extension libraries (see above).

Indexing

ANALYTICS library allows using **indexing** in expressions. Indexing is a method of access to the structured data elements. An example of structured data is array. Each array's element has the unique index by which the element value is accessed. By syntax rules data indexes are written in square brackets '[]'. For an example, the i-th array element can be written as '**A[i]**'. For multiple indexes, each index must be written in different brackets. For an example, a matrix

---

[2] 'Special' operators have no predefined meaning for common data types (real, complex and so on) and can be used for special needs with program specific types. Note that the syntax rules cannot be changed for the operators in any case.

element can be written as '**M[i][j]**'. Such syntax allows **slicing** implementation. Slicing allows to get a part of structured data. Slicing is implemented via index omitting. Let we have a matrix (two-dimensional array) '**M**'. Then '**M[i][]**' is the i-th matrix' row and '**M[][j]**' is the j-th matrix' column. The 'trail' indexes can be omitted with their brackets. Thus, the matrix' row can be written as '**M[i]**' (which is equivalent to '**M[i][]**' of the previous example).

General indexing rules:
- indexing can only be applied to variables, the variables must implement indexing interface (see below);
- many indexes allowed, each index must be enclosed in square braces '**[]**';
- all index expressions must return values of real type only, the values must be (almost) integer;
- some indexes can be omitted that means slicing, slicing can only be applied to variables which implement slicing interface (see below);

ANALYTICS library contains predefined array variable types. Array variables can contain data of any type. Array variables have default indexing and slicing implementation for arrays up to third dimension.

Syntax summary

- Any expression can contain literals, variables, operators, functions and indexing.
- Literals can be real numbers in simple and exponent form and complex numbers.
- Variable names can include alpha-numerics and underscore symbol only.
- Only predefined (see above) operators can be used, new operators cannot be defined.
- Operations are performed in order of operator precedence. Operations order can be changed using parantheses '()'.
- Functions have parameters and arguments. Parameters are enclosed in braces '**{}**', arguments are enclosed in parantheses '()'. Parameters and arguments are separated by spaces ' '.
- Indexing can only be applied to variables (those implement indexing interface. Indexes are enclosed in brackets '[]' (each index in separate brackets). Slicing of indexed data can be implemented by omitting some indexes.

Some examples of syntactically correct formulae:
```
'sin(b)^2+cos(b)^2'                       - Pythagorean trigonometric identity formula.
'sin(a)*cos(b)+cos(a)*sin(b)'             - Sine of angle sum formula.
'log{c}(a)+log{c}(b)'                     - Logarithm of product formula.
'A[n]*r^n*sin(n*a)+B[n]*r^-n*cos(n*a)'    - Laplace's equation solution in polar coordinates.
'A[n]*sinh(n*Pi/L*(x-a))*sin(n*Pi/L*(y-b))' - Laplace's equation solution in Cartesian system.
```

2. Class Hierarchy

This part explains the base class hierarchy of ANALYTICS library. It is not needed to understand the hierarchy to use the library for common analytical calculations. The knowledge is only useful for extending library (introducing new function, overloading operators and so on).

The main concept of ANALYTICS library is easy external extensibility. That is the library has the complete core which is not changed and new functionality can be implemented by (only) attaching external libraries. So there is strongly structured class hierarchy to implement this concept.

Variable classes

Variable is a name with associated value (of some type). The base abstract class **Variable** is intended to use this concept inside the program. The main properties of the class are: **Name**, **Value** and **ValueType**. There are two abstract classes, inherited from the **Variable** class. **ScalarVariable** class can contain one value. **IndexedVariable** class introduces interface for indexed value, that can contain another values accessed by indexes. The following classes inherited from **ScalarVariable**: **RealVariable** (contains real value), **BooleanVariable** (contains boolean value), **ObjectVariable** (contains value of any type). **BaseArrayVariable** class is inherited from **IndexedVariable** and is abstract class for variables those contain arrays (of any dimension). Three abstract classes **ArrayVariable**, **MatrixVariable** and **BlockVariable** realize interfaces for one-, two- and three-dimensional arrays accordingly. And finally, concrete classes of real and object arrays up to the third dimension are realized.

All concrete classes, introduced in this hierarchy, are fully functional. That is they realize all functionality to use them in calculations. All array variables realize indexing and slicing interfaces.

There are other variable types realized in extension libraries (complex variables, 3D vector and tensor variables and so on). They can be used in the same way as variables built in the ANALYTICS core library.
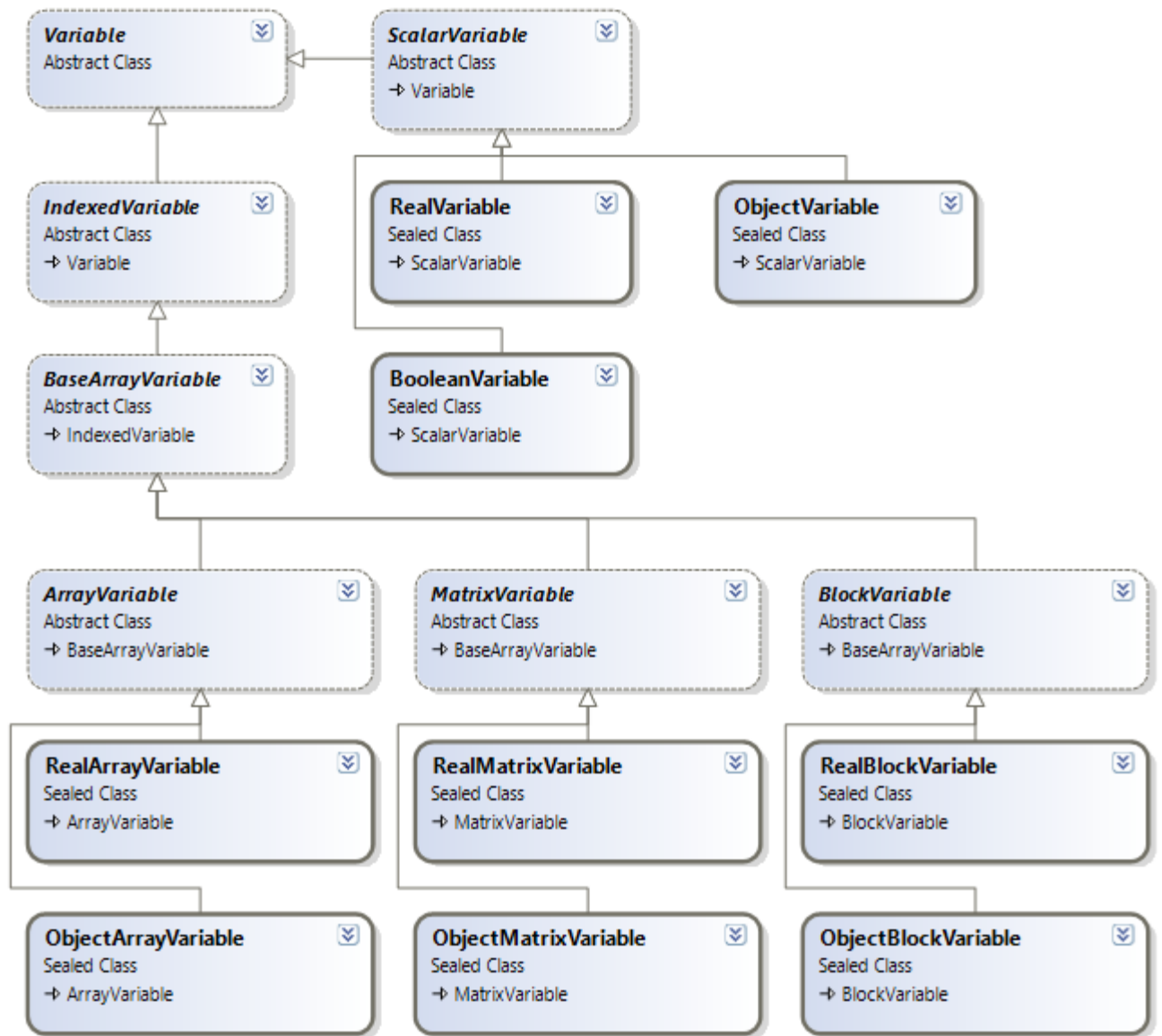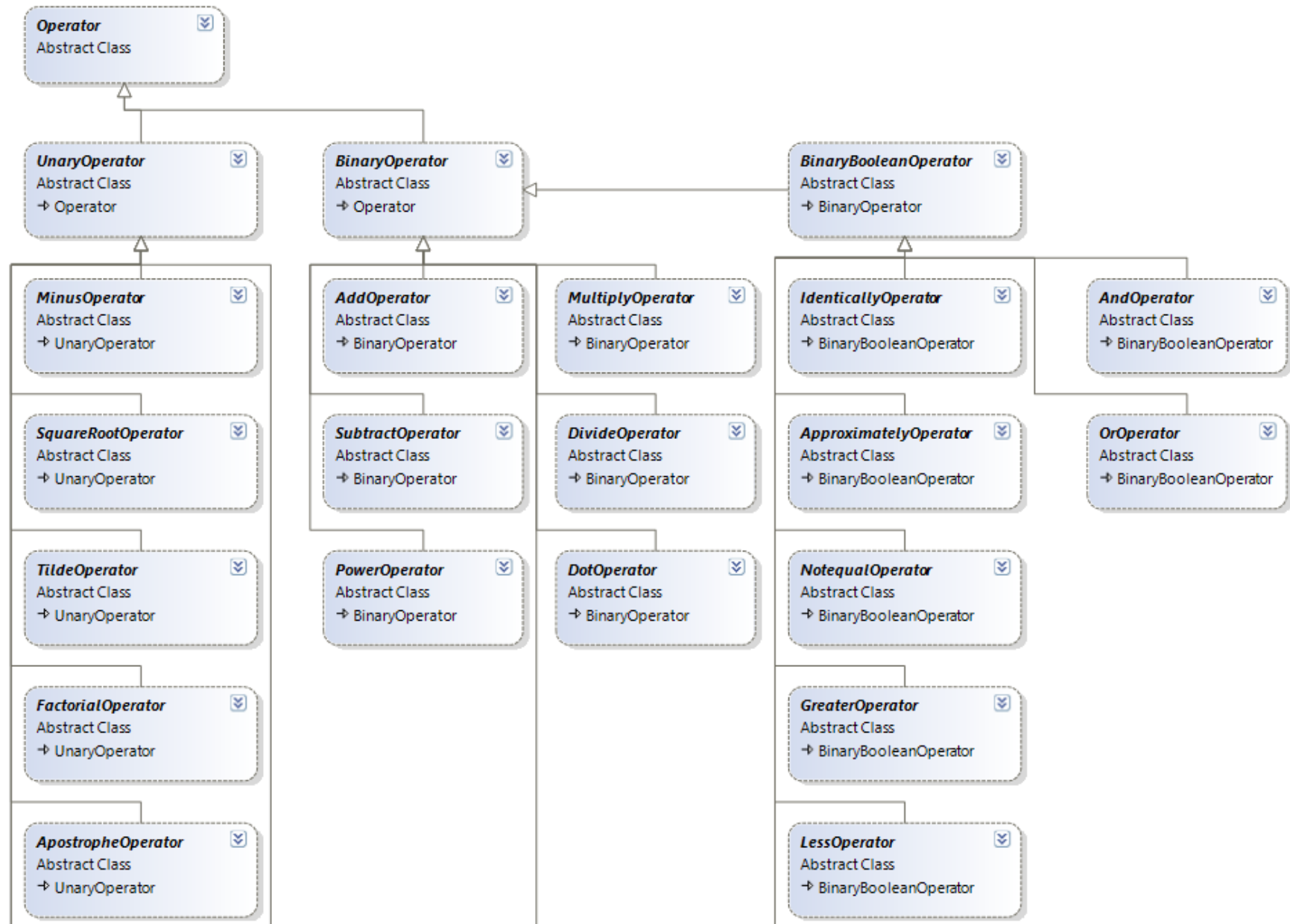
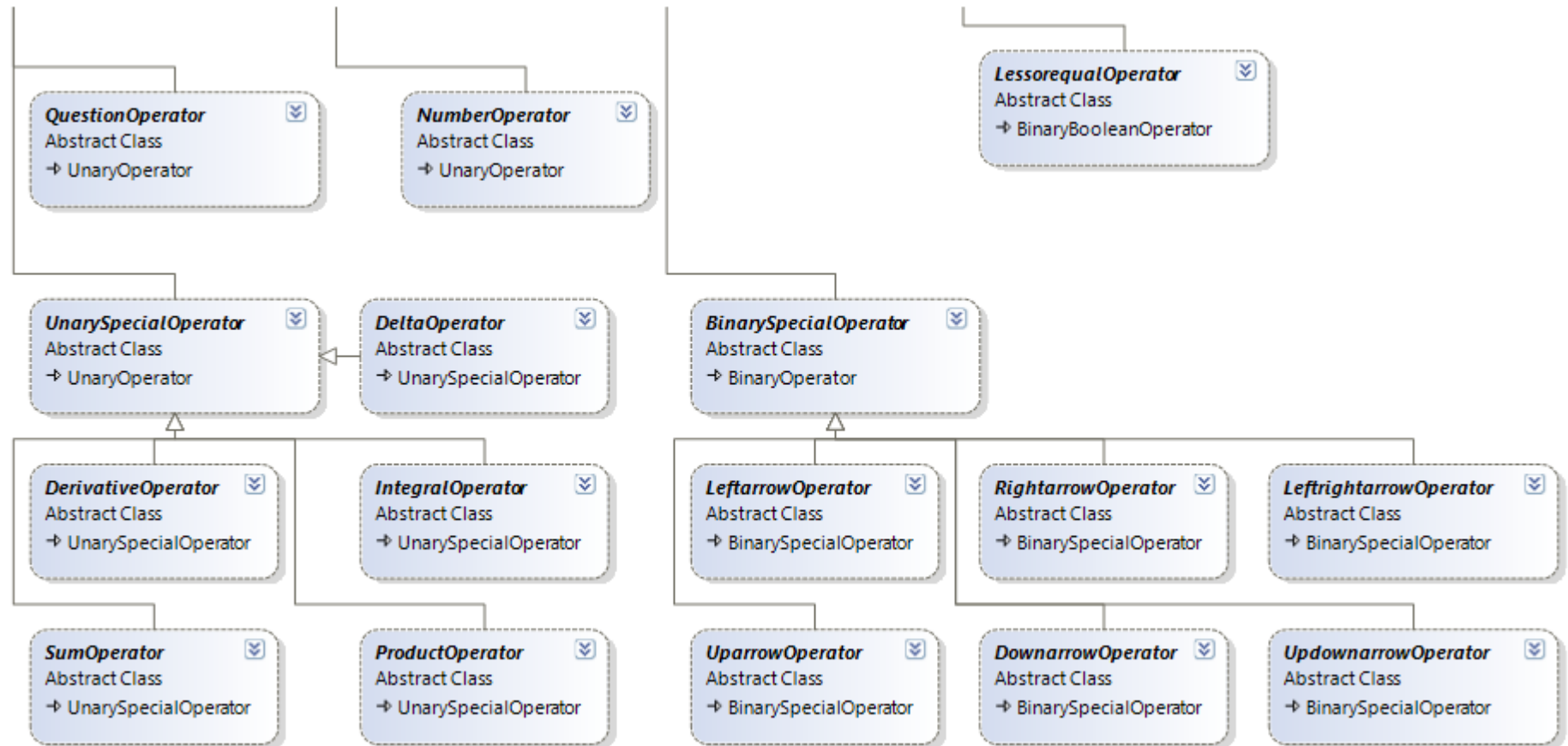Picture 2.1. Variable class hierarchy diagram.

Operator classes

Base abstract class **Operator** is intended to use the concept of operator (symbol for some operation with operand values) inside the program. This class has properties and functions to define the symbol of the operator, the operation result type and the method to calculate value. In accordance with operator types two abstract classes are inherited from the base one: **UnaryOperator** and **BinaryOperator**. These classes introduce interface to define operand type(s). More specific classes, inherited from the last ones, define the concrete operator type and symbol ('+', '^' and so on).

The concrete classes, those realize the complete functionality (define the operand types and perform operations on the operand's values) are not shown on the diagram. There are all realized operators for Real operands. Operators for other operand types (complex, 3D vectors and tensors and so on) realized in ANALYTICS extension libraries.

There are also **generic** analogues for all operator classes. For an example, generic analogue for **AddOperator** is **GenericAddOperator** class. These generic classes can be also used to derive new operator classes. The generic form of these classes simplifies inheritance because they already contain implementation of some methods (see operator overloading below).

Operator
Abstract Class

UnaryOperator
Abstract Class
↗ Operator

BinaryOperator
Abstract Class
↗ Operator

BinaryBooleanOperator
Abstract Class
↗ BinaryOperator

MinusOperator
Abstract Class
↗ UnaryOperator

AddOperator
Abstract Class
↗ BinaryOperator

MultiplyOperator
Abstract Class
↗ BinaryOperator

IdenticallyOperator
Abstract Class
↗ BinaryBooleanOperator

AndOperator
Abstract Class
↗ BinaryBooleanOperator

SquareRootOperator
Abstract Class
↗ UnaryOperator

SubtractOperator
Abstract Class
↗ BinaryOperator

DivideOperator
Abstract Class
↗ BinaryOperator

ApproximatelyOperator
Abstract Class
↗ BinaryBooleanOperator

OrOperator
Abstract Class
↗ BinaryBooleanOperator

TildeOperator
Abstract Class
↗ UnaryOperator

PowerOperator
Abstract Class
↗ BinaryOperator

DotOperator
Abstract Class
↗ BinaryOperator

NotequalOperator
Abstract Class
↗ BinaryBooleanOperator

FactorialOperator
Abstract Class
↗ UnaryOperator

GreaterOperator
Abstract Class
↗ BinaryBooleanOperator

ApostropheOperator
Abstract Class
↗ UnaryOperator

LessOperator
Abstract Class
↗ BinaryBooleanOperator

QuestionOperator
Abstract Class
→ UnaryOperator

NumberOperator
Abstract Class
→ UnaryOperator

LessorequalOperator
Abstract Class
→ BinaryBooleanOperator

UnarySpecialOperator
Abstract Class
→ UnaryOperator

DeltaOperator
Abstract Class
→ UnarySpecialOperator

BinarySpecialOperator
Abstract Class
→ BinaryOperator

DerivativeOperator
Abstract Class
→ UnarySpecialOperator

IntegralOperator
Abstract Class
→ UnarySpecialOperator

LeftarrowOperator
Abstract Class
→ BinarySpecialOperator

RightarrowOperator
Abstract Class
→ BinarySpecialOperator

LeftrightarrowOperator
Abstract Class
→ BinarySpecialOperator

SumOperator
Abstract Class
→ UnarySpecialOperator

ProductOperator
Abstract Class
→ UnarySpecialOperator

UparrowOperator
Abstract Class
→ BinarySpecialOperator

DownarrowOperator
Abstract Class
→ BinarySpecialOperator

UpdownarrowOperator
Abstract Class
→ BinarySpecialOperator

Picture 2.2. Operator class hierarchy diagram.

Function classes

The base abstract class **Function** implements the concept of function. The class has interface to define the name, the count and type of parameters and arguments, the type of returned value and the method to make operation on the data values. The class **MonotypeFunction** is directly inherited from the **Function** and implements the concept of function such that all parameters and arguments are of the same type as the returned value. At the next level of hierarchy all functions are divided into **Elementary** and **Special**. This division is to correspond with mathematics, not for programming convenience.
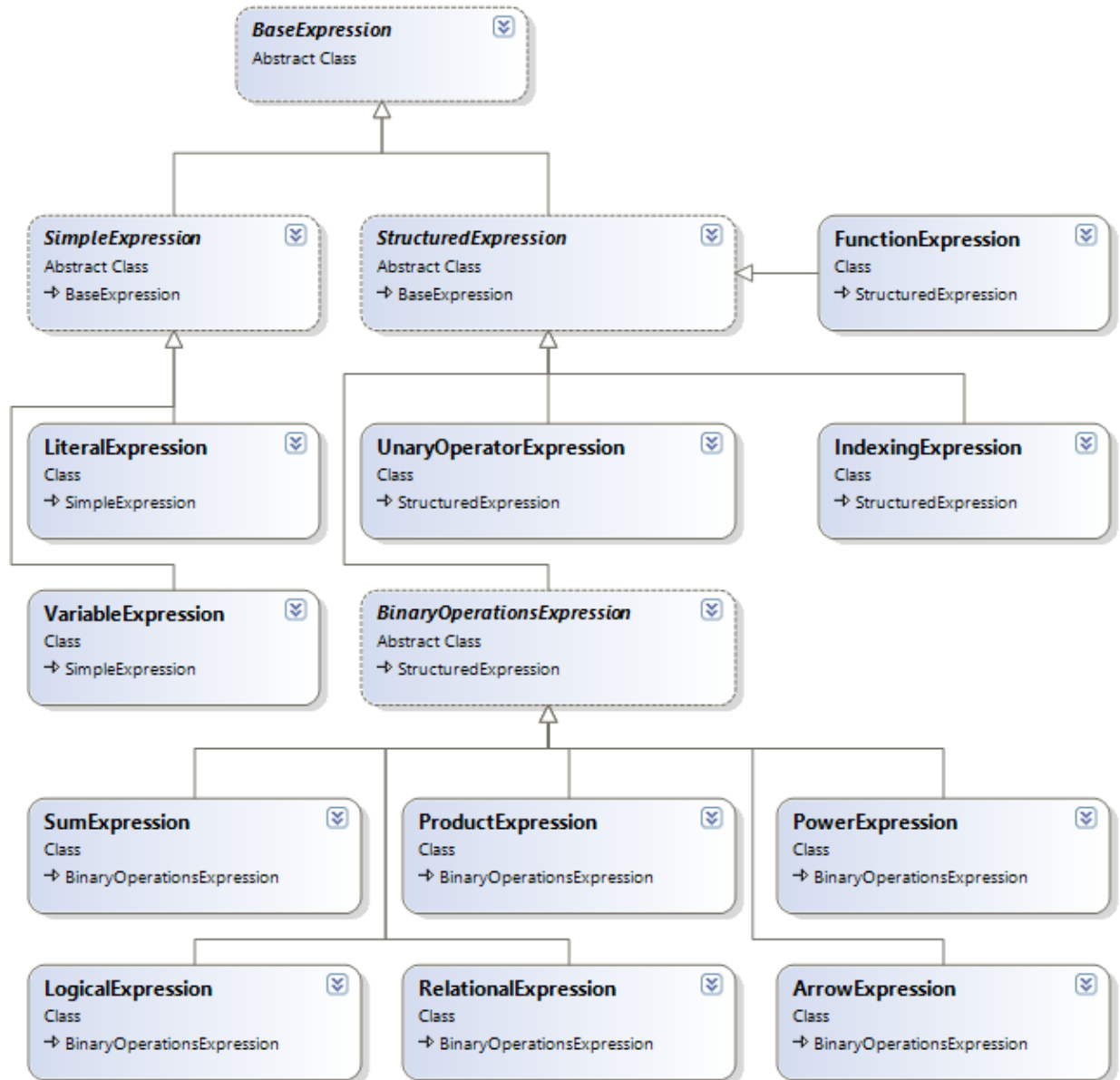

Picture 2.3. Function class hierarchy diagram.

Further, each of classes is divided into Real and Complex subclasses (functions with Real and Complex arguments accordingly) and finally all classes are divided into simple and Parametric functions. Simple functions have one argument, parametric – one parameter and one argument.

The concrete classes those realize the calculation are not shown on the diagram. There are many realized elementary and special functions of Real arguments – algebraic, trigonometric, inverse trigonometric, exponential, logarithmic, hyperbolic, inverse hyperbolic. Functions for other argument types (complex, 3D vectors and tensors and so on) implemented in ANALYTICS extension libraries (see above).

Expression classes

Expression classes are intended to represent data inside core algorithms of the ANALYTICS library. The base abstract class for all expressions is *BaseExpression.*

Picture 2.4. Expression class hierarchy diagram.

This class defines abstract interface for all expressions and implements some static methods to manipulate with expressions (build expressions form strings, simplify expression list and so on). All other expressions are divided into **simple** (do not contain other expressions) and **structured** (do contain other expressions) ones. Simple expressions are *LiteralExpression* and *VariableExpression*. Structured expressions include *FunctionExpression*, *IndexingExpression*, *UnaryOperatorExpression* and *BinaryOperationsExpression* (*LogicalExpression, RelationalExpression, SumExpression*, *ProductExpression*, *PowerExpression, ArrowExpression*). All expression classes realize methods to manipulate with them: build expressions from strings, simplify expressions, reconstruct expressions (convert them back to the string), building new expressions from existing and others.

It is not obligatory to understand expression class functionality to use the ANALYTICS library. It is only needed to extend the functionality of the library in the sense of analytical derivative calculation (see below).

3. Working with ANALYTICS

The main functionality of ANALYTICS library (evaluation of mathematical expressions) is realized in **Translator** class. **Translator** class encapsulates a set of variables and a set of operations (registered operators and functions). Thus **Translator** can parse string expressions and calculate their values for current variable values.

Translator is not a static class because various translators can have different operation and variable sets. So to use **Translator** functionality an instance of the class must be created. Hereinafter it is supposed that the instance 'translator' of **Translator** class has been created

```
Translator translator = new Translator();
```

Working with variables

Translator class has complete interface to add and remove variables and to change their values.

To add variables use **Add** method. This method has many overloads to add variables of different types. The following code examples demonstrate the process of variable addition:

- adding Real variable
```
string name = "a";
double v = 1.0;
translator.Add(name, v);
```

- adding Complex variable
```
string name = "x";
Complex z = new Complex(1.0, -2.0);
Variable v = new ComplexVariable(name, z);
translator.Add(v);
```

- adding Real array variable
```
string name = "A";
double[] a = new double[] { 0.0, 1.0, 2.0, 3.0 };
Variable v = new RealArrayVariable(name, a);
translator.Add(v);
```

- adding variable of any type
```
string name = "A";
Complex[] a = new Complex[] { new Complex(0,0), new Complex(1,-1), new Complex(-2,2) };
Variable v = new ComplexArrayVariable(name, a);
translator.Add(v);
```

There are two ways to change variable values.
1. If the direct reference to a variable is available, the value can be changed using **Value** property of the variable class.
```
string name = "a";
double a = 1.0;
Variable v = new RealVariable(name, a);
translator.Add(v);
// some code here...
v.Value = (double)2.0;
```

2. If there is no direct reference to a variable, it can be got by using **Get** method of the **Translator** class (the name of variable or its index can be used).
```
Variable x = translator.Get("x");
x.Value = (double)3.0;
// some code here...
Variable y = translator.Get(1);
y.Value = (double)4.0;
```

NOTE about changing variable values: the type of the variable value cannot be changed. In spite of the property **Value** of the **Variable** class is of type **Object**, when setting variable value its type must be the same as current **ValueType**. The value type is determined at the variable creation and cannot be changed during variable lifetime.

The **Delete** method of **Translator** class can be used to remove variables. The variable can be removed from the translator instance by name or its index in the translator's table. The **DeleteAll** method removes all variables. The following code demonstrates the process of variable deleting.

```
translator.Delete("x");
// some code here...
translator.Delete(2);
// some code here...
translator.DeleteAll();
```

Calculating formula values

In the simplest case the **Calculate** method of **Translator** class can be used to calculate formula value. The following code demonstrates the method using

```csharp
string f1 = "sin(a)^2+cos(a)^2";
double r = (double)translator.Calculate(f1);
// some code here...
string f2 = "2*exp(z)-I*sin(z/3)";
Complex c = (Complex)translator.Calculate(f2);
```

In the code above it is supposed that there are variables "a" and "z" in translator's variable set and "a" is of Real type and "z" is a complex one. Note, that the return type of the **Calculate** method is **Object** and it can return the value of any type depending on the formula contents. So, the returned value must be directly casted to the type and the type must be known.

The **Calculate** method is rather slow. This is because it parses string expression and creates internal structure to calculate result value. Parsing methods are not optimized for speed, they are optimized for strong object oriented structuring and easy extensibility. Thus the usage of **Calculate** method is only recommended for single formula evaluation.

Another case of formula evaluation comes from the need to calculate one expression for various variable values. This can be done in such a way:

- creating **Formula** object from string expression;
- changing variable values;
- calculation formula values for current variable values.

Formula is a class intended for internal program representation of parsed mathematical expressions. The following code demonstrates calculation of the table of function values for various argument values by the above algorithm.

```csharp
string s = "2*(sin(x)+cos(x))";
Formula f = translator.BuildFormula(s); // parsing string expression
Variable x = translator.Get("x");
double v = 0.0;
double[] ax = new double[101];
double[] ay = new double[101];
for (int i = 0; i <= 100; i++)
{
        ax[i] = v;
        x.Value = v; // setting new variable value
        ay[i] = (double)f.Calculate(); // calculating formula value for current x value
        v += 0.01;
}
```

In the code, the string expression is parsed only once by the **BuildFormula** function which returns the **Formula** object. When the **Formula** instance created, its value can be calculated many times for various 'x' variable values.

Checking expression syntax

In all above code examples it was supposed that the string expressions were syntactically correct. End user applications, of course, must check the syntax correctness of user defined expressions. The **Translator** class provides methods to check expressions before calculation.

The syntax correctness must be checked in three steps. The first, syntax rules, those do not need the expression to be parsed, must be checked. For an example, the parantheses in a mathematical expression must be pairwise and it can be checked without expression decomposition. Such syntax rules must be checked before any calculation by **Translator**'s **CheckSyntax** method. This function returns true, if all rules are fulfilled and throws an exception if not.

The second step is to check, that a string can be decomposed into known expression types. For an example, the string 'sin(x+1)' can be decomposed as a function with name 'sin' and one argument 'x+1' which is itself the known expression – the sum of constant '1' and variable with name 'x'. This step does not need to know that the function 'sin' is defined or the variable 'x' exists.

The third step checks that all elements in decomposed expression are defined. It means that all variables must exist. Moreover, this step needs to know the types of expression results to check that all operations in the expression can be performed. For an example, in given expression, if variable 'x' is Real, the operator '+' must be defined for Real operands and the function 'sin' with one Real argument must be registered.

The second and the third steps of syntax checking are implemented in **Translator**'s **BuildFormula** method. This method returns built **Formula** object, if the string expression is correct, and throws an exception if not.

The following example code demonstrates common syntax checking algorithm:

```csharp
string s = "2*(sin(x)+cos(x))";
try
{
        // the first step of syntax checking
        if (translator.CheckSyntax(s))
        {
                // the second and third steps of syntax checking
                Formula f = translator.BuildFormula(s);
                if (f != null)
                {
                        // here the formula calculation code
                        // using f instance.
                }
        }
```

```
        }
        catch (Exception ex)
        {
                // here the exception handling code.
        }
```

Analytical derivative calculation

The ANALYTICS library allows to calculate derivatives of expressions. The **Translator** class (see above) contains the following method:

```
public string Derivative(string formula, string vName)
```

This method calculates analytical derivative of the *formula* by variable *vName*. The result is string representing the derivative expression.

Here are some example codes of analytical derivative calculation:

```
string formula;
string derivative;
// #1
formula = "A*ln(x)*sin(2*x)";
derivative = translator.Derivative(formula, "x");
// derivative = (1/x)*(A*sin(2*x))+(cos(2*x)*2)*(A*ln(x))
// #2
formula = "e^(1-2*x)";
derivative = translator.Derivative(formula, "x");
// derivative = e^(1-2*x)*(-2)
// #3
formula = "(x+1)^(x-1)";
derivative = translator.Derivative(formula, "x");
// derivative = ln(x+1)*(x+1)^(x-1)+(x-1)*(x+1)^((x-1)-1)
```

The examples show that the calculation is very simple.

Notes about derivative calculations:

1. The *formula* parameter in **Derivative** method must be syntactically correct. The correctness can be checked by **Translator**'s **CheckSyntax** method.
2. There is no need variable with the name *vName* (or any other) to be existing. The **Derivative** method manipulates variable names only, not their values.
3. The derivative method does some simplification after calculation (removes zero values from sums, unit multipliers from products and so on). But some results can be not 'beautiful to see'. They can contain some superfluous braces and other 'artifacts'. It is because the ANALYTICS library is not computer algebra system. The derivative calculation result is intended to use in the subsequent calculations (inside the program), not to show it to user.
4. Not all operators supported for derivative calculations. For an example, '!' operator not supported, because the derivation rule is not defined for it. It is not possible to introduce new derivation rules for operators (without rewriting core algorithms).
5. Most of standard transcendental functions supported for derivative calculation. Derivation rule can be defined for any function (see below).

Explicit string expressions manipulation

The ANALYTICS library allows 'explicit' manipulations with 'string expressions'. Let there are two strings inside the program, containing mathematical expressions '**1-2*x**' and '**sin(x)+2**'. And let there is need to get the expression which is the multiplication of the given ones. This task can be done just manipulating with C# strings. For the above example strings the following steps must be done: enclose the first expression into parentheses, enclose the second expression into parentheses and finally concatenate the result strings with the '*' operator. The algorithm seems to be simple, but it becomes more and more complicated with the number of operations increase.

The **Expression** structure (do not confuse with the **BaseExpression** from class hierarchy) simplifies such string manipulations. This structure type is just 'simple string wrapper' (pic. 3.1). It contains explicit conversion operators – from string and to string and overloaded operators implementing algebraic operations '**+**', '**-**', '*****', '**/**', '**^**' (power). This implementation allows manipulating string expressions in 'natural' manner.

Consider an example of using the **Expression** structure for the following task: realize 3D 'analytical' vector. This vector must contain three string components, all components must be mathematical expressions and all vector manipulations (sum, vector product, length and so on) must be analytical.
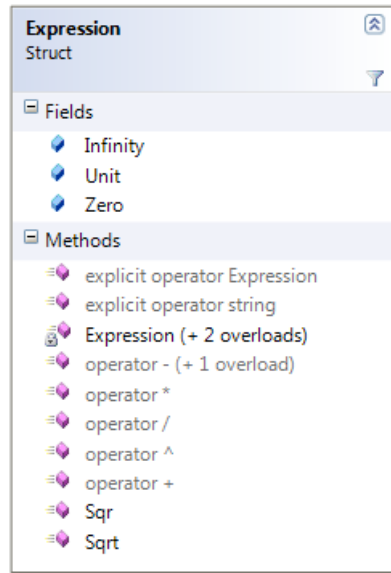
Picture 3.1. Expression class.

     The following code is the (partial) implementation of 'analytical' 3D vector:

```csharp
public struct StringVector
{
    #region Data members
    private Expression e1;
    private Expression e2;
    private Expression e3;
    #endregion Data members

    /// <summary>
    /// Constructor.
    /// </summary>
    /// <param name="x1"></param>
    /// <param name="x2"></param>
    /// <param name="x3"></param>
    public StringVector(string x1, string x2, string x3)
    {
        e1 = (Expression)x1;
        e2 = (Expression)x2;
        e3 = (Expression)x3;
    }

    /// <summary>
    /// Dot Product
    /// </summary>
    /// <param name="v1"></param>
    /// <param name="v2"></param>
    /// <returns></returns>
    public static string Dot(StringVector v1, StringVector v2)
    {
        return (string)(v1.e1 * v2.e1 + v1.e2 * v2.e2 + v1.e3 * v2.e3);
    }

    /// <summary>
    /// Cross Product
    /// </summary>
    /// <param name="v1"></param>
    /// <param name="v2"></param>
    /// <returns></returns>
    public static StringVector operator *(StringVector v1, StringVector v2)
    {
        return new StringVector(
            (v1.e2 * v2.e3 - v1.e3 * v2.e2),
            (v1.e3 * v2.e1 - v1.e1 * v2.e3),
            (v1.e1 * v2.e2 - v1.e2 * v2.e1)
        );
    }
}
```

     The **StringVector** structure contains three data members e1, e2, e3 of type **Expression** to encapsulate data for three vector components. The constructor takes three string arguments, which are converted to **Expression** data using explicit operator.

Main advantages of using **_Expression_** structures are demonstrated in **_Dot_** and **_Cross_** methods. The calculation codes of vector dot and cross products are **similar** to if the vectors have **_double_** components.

Consider the result of using StringVector structure:

```
StringVector x1 = new StringVector("sin(u)", "cos(v)", "0");
StringVector x2 = new StringVector("cos(u)", "-sin(v)", "0");
StringVector r =  x1 * x2;
```

The result vector r will contain three expression components '0', '0' and 'sin(u)*(-sin(v))-cos(v)*cos(u))'.

NOTE about using **_Expression_** structure: all string values to manipulate with **_Expression_** structure must be syntactically correct; else, the exception will be thrown. The syntax can be checked by **_Translator_**'s **_CheckSyntax_** method (see above).

4. Extending ANALYTICS

The ANALYTICS library provides complete core for parsing and calculating mathematical expressions. It also contains many predefined 'standard' functions – trigonometric, hyperbolic and so on and realized operators for real arguments. So, the library can be used without modification to provide in the end user program the interface for user to input data in the form of mathematical expressions.

Another goal of the library is to provide the possibility of mathematical calculations with the program specific data. For an example, let there is a program for signal processing. And the program must allow the end user to make various operations with signals – add two signals, multiply a signal by real values, calculate exponents and logarithms of signals and so on. This can be done using the ANALYTICS library. The core algorithms of ANALYTICS library can be used to calculate expressions containing signals as variables. The only thing required is to provide operations, defined for signals (because they are program specific data).

The ANALYTICS library is built by such technology that allows easily introduce operations with program specific data without changing core algorithms. All data operations (operators, functions) are presented inside the library as classes. To add an operation with program specific data a descendant of some class must be implemented. This class will be automatically found and used to calculate expression results.

The next part explains implementation of classes to define operations with program specific data.

Overloading operators

To overload an operator for program specific data operand types the descendant of one of the defined operator classes must be implemented. There are base abstract classes for all defined operators (see class hierarchy): AddOperator, SubtractOperator and so on. The following code demonstrates the operator overloading for arrays of real numbers:

```csharp
/// <summary>
/// (Double Array) * (Double) = (Double Array)
/// </summary>
public sealed class ArrayRealMultiply : MultiplyOperator
{
    protected override Type GetOperand1Type()
    {
        return typeof(double[]);
    }

    protected override Type GetOperand2Type()
    {
        return typeof(double);
    }

    protected override Type GetReturnType()
    {
        return typeof(double[]);
    }

    protected override object Operation(object operand1, object operand2)
    {
        if (operand1 == null) return null;

        double[] a = (double[])operand1;
        double r = (double)operand2;

        int l = a.Length;
        double[] result = new double[l];

        for (int i = 0; i < l; i++)
        {
            result[i] = a[i] * r;
        }

        return result;
    }
}
```

The **ArrayRealMultiply** class is inherited from **MultiplyOperator**, this means it overloads "*" operator. The class overrides **GetOperand1Type**, **GetOperand2Type** and **GetReturnType** methods and defines that the first operand is of type array of doubles and the second is of type double, the operation's result is double array. Another overridden method is **Operation** – this method implements the action that must be performed on operands.

The following example demonstrates overloading of "+" operator for arrays of real values:

```csharp
/// <summary>
/// (Double Array) + (Double Array) = (Double Array)
/// </summary>
public sealed class ArrayAdd : AddOperator
{
    protected override Type GetOperand1Type()
    {
        return typeof(double[]);
    }

    protected override Type GetOperand2Type()
    {
        return typeof(double[]);
    }

    protected override Type GetReturnType()
    {
        return typeof(double[]);
    }

    protected override object Operation(object operand1, object operand2)
    {
        if (operand1 == null || operand2 == null) return null;

        double[] a1 = (double[])operand1;
        double[] a2 = (double[])operand2;

        int l = a1.Length;
        double[] result = new double[l];

        for (int i = 0; i < l; i++)
        {
            result[i] = a1[i] + a2[i];
        }

        return result;
    }
}
```

The classes of operators will be automatically found by ANALYTICS library and used to implement operations with operands of defined types.

The same result, as in two previous code examples, can be achieved by using generic analogues of base operators. The following code demonstrates this case:

```csharp
/// <summary>
/// (Double Array) * (Double) = (Double Array)
/// </summary>
public sealed class ArrayRealMultiply : GenericMultiplyOperator<double[], double, double[]>
{
    protected override double[] TypedOperation(double[] operand1, double operand2)
    {
        if (operand1 == null) return null;

        int l = operand1.Length;
        double[] result = new double[l];

        for (int i = 0; i < l; i++)
        {
            result[i] = operand1[i] * operand2;
        }

        return result;
    }
}

/// <summary>
/// (Double Array) + (Double Array) = (Double Array)
/// </summary>
public sealed class ArrayAdd : GenericAddOperator<double[], double[], double[]>
{
    protected override double[] TypedOperation(double[] operand1, double[] operand2)
    {
```

```
        if (operand1 == null || operand2 == null) return null;

        int l = operand1.Length;
        double[] result = new double[l];

        for (int i = 0; i < l; i++)
        {
            result[i] = operand1[i] + operand2[i];
        }

        return result;
    }
}
```

Using generic base operators is "shorter" because only one method (*TypedOperation*) must be overridden. Operand and return types are defined as parameters of the generic class the operator is inherited from. Both inheritance cases, generic and not generic, absolutely equivalent for ANALYTICS core algorithms.

Explicitly overloaded operators

The operator overloading in ANALYTICS library is intended to implement operations on program specific type data. Often, for such data, some operators are already overridden "inside" the program (that means C# operator overloading). And these "explicitly overloaded operators" can be used to implement operations on such data inside ANALYTICS core algorithms. In other words, if there is an operator "explicitly overloaded" it will be automatically found and used to calculate result for the suitable operation. For an example, as *Complex* type overloads math operators, such actions as addition, subtraction and so on can be performed for complex numbers without deriving new operator classes.

There are some constraints for using "explicitly overloaded" operators:
1. Not all operators can be "explicitly overloaded", because not all ANALYTICS operators have analogous C# operators or their "meaning" can be ambiguous ("**^**" for an example). So, only following operators can be "explicitly overloaded": "**+**", "**-**" (unary and binary), "*****", "**/**", '**≡**' (==)[3], '**≠**' (!=), '**>**', '**<**', '**≥**' (>=), '**≤**' (<=), '**&**', '****' (|), '**¬**' (!).
2. "Implicit" overloading (deriving new operator classes) has higher priority. That is if there are both "explicit" and "implicit" operators – the last will be used to calculate operation result.

Both overloading methods can be used in combination. "Implicit" overloading (using new operator classes) can be used as for complimenting "explicit" method, as for overriding its behavior.

Introducing new functions

Implementing functions for program specific data is more general way of extending library functionality. A Function can have any valid name, any number of arguments of different types and any return type. To introduce new function a descendant of one of abstract function classes must be implemented. The choice of a base class is wider, than for operators (see class hierarchy). For common argument types (real or complex), one of the predefined abstract classes can be used as ancestor class: *RealElementaryFunction*, *RealParametricElementaryFunction*, *ComplexElementaryFunction*, *ComplexParametricElementaryFunction*. As an example, the code of class, implementing sine function of complex argument, is listed below:

```
/// <summary>
/// Sine function of Complex Argument
/// </summary>
public sealed class SineComplex : ComplexElementaryFunction
{
    protected override string GetName()
    {
        return "sin";
    }

    protected override Complex Func(Complex x)
    {
        return Complex.Sin(x);
    }
}
```

The class is inherited from ComplexElementaryFunction because it implements a function with one argument of Complex type. The only methods overridden are GetName and Func. The former defines the function name (used in expressions) and the later implements the function operation itself. Another example demonstrates the logarithm function implementation of complex argument and base:

```
/// <summary>
/// Logarithm function of Complex argument
/// (NOTE: the Base of logarithm is the Parameter of the function)
/// </summary>
```

---

[3] C# analogue operators shown in parentheses.

```csharp
public sealed class LogarithmComplex : ComplexParametricElementaryFunction
{
    protected override string GetName()
    {
        return "log";
    }

    protected override Complex Func(Complex parameter, Complex argument)
    {
        return Complex.Log(argument)/Complex.Log(parameter);
    }
}
```

The class is inherited from ComplexParametricElementaryFunction because it implements a function with one parameter and one argument of Complex type.

In general case, to introduce new function the class must be inherited from the base abstract class **Function**. All abstract methods of the class must be overridden. The methods must define the number and types of parameters and arguments and the function return type. The following code example demonstrates the implementation of Min function that calculates the minimum value in an array of real numbers:

```csharp
/// <summary>
/// Min array element
/// </summary>
public sealed class MinArray : Function
{
    protected override string GetName()
    {
        return "Min";
    }

    protected override int GetArgumentCount()
    {
        return 1;
    }

    protected override Type[] GetArgumentTypes()
    {
        return new Type[] { typeof(double[]) };
    }

    protected override int GetParameterCount()
    {
        return 0;
    }

    protected override Type[] GetParameterTypes()
    {
        return null;
    }

    protected override Type GetResultType()
    {
        return typeof(double);
    }

    protected override object DoCalculate(object[] parameters, object[] arguments)
    {
        double[] a = (double[])arguments[0];
        if (a == null || a.Length == 0) return double.NaN;

        double result = a[0];
        int l = a.Length;
        for (int i = 1; i < l; i++)
            if (a[i] < result) result = a[i];

        return result;
    }
}
```

The function has one argument of type real array and returns real value.

A little shorter way to introduce new function for the specific types is using base generic function classes. These classes use generic parameters to define the types of arguments. Thus, only Name and calculation algorithm must be provided to define new function class. The following code demonstrates the previous example function implementation using base generic class:

```csharp
/// <summary>
/// Min array element
/// </summary>
```

```csharp
public sealed class MinArray : GenericSimpleFunction<double[], double>
{
    protected override string GetName()
    {
        return "Min";
    }

    protected override double Func(double[] a)
    {
        if (a == null || a.Length == 0) return double.NaN;

        double result = a[0];
        int l = a.Length;
        for (int i = 1; i < l; i++)
            if (a[i] < result) result = a[i];

        return result;
    }
}
```

The class is inherited from **GenericSimpleFunction**, which means it has one argument and zero parameters. The argument type is real array and the function's result is real which is provided by the first and the second generic parameters accordingly.

The ANALYTICS library contains base generic function classes to implement functions with up to two parameters and two arguments of any type.

NOTE about functions: there can be many functions with the same name but different count or/and type of arguments (parameters).

Implementing indexing

Indexing can be applied for variables only. To implement indexing for program specific data new variable class must be inherited from the abstract class **IndexedVariable** (or from one of its descendants). The inherited class must override methods defining the number of data indexes, data access methods and slicing implementation.

As an example of indexing implementation, the (partial) code of standard **MatrixVariable** class is listed below:

```csharp
/// <summary>
/// Base abstract class for all matrix variables.
/// NOTE: Slicing is implemented.
/// </summary>
public abstract class MatrixVariable : BaseArrayVariable
{
    /// <summary>
    /// 2 indexes
    /// </summary>
    /// <returns></returns>
    protected override sealed int GetIndexCount()
    {
        return 2;
    }

    /// <summary>
    /// Slicing is implemented for Matrix variables
    /// </summary>
    /// <returns></returns>
    protected override bool GetSlicingImplemented()
    {
        return true;
    }

    /// <summary>
    /// Sliced item is array of BaseType
    /// </summary>
    /// <param name="indexes"></param>
    /// <returns></returns>
    public override Type GetItemType(int[] indexes)
    {
        if ( (indexes[0] >= 0 && indexes[1] < 0)
            || (indexes[0] < 0 && indexes[1] >=0) )
        {
            return BaseType.MakeArrayType();
        }

        return BaseType;
    }

    /// <summary>
    /// Implements Array Slicing
    /// </summary>
```

```csharp
/// <param name="indexes"></param>
/// <returns></returns>
public override object GetItemValue(int[] indexes)
{
    // Row
    if (indexes[0] >= 0 && indexes[1] < 0)
    {
        int len = ColumnCount;
        Array result = Array.CreateInstance(BaseType, len);
        int[] row = new int[] { indexes[0], 0 };

        for (int j = 0; j < len; j++)
        {
            row[1] = j;
            object x = data.GetValue(row);
            result.SetValue(x, j);
        }

        return result;
    }
    else
    {
        // Column
        if (indexes[0] < 0 && indexes[1] >= 0)
        {
            int len = RowCount;
            Array result = Array.CreateInstance(BaseType, len);
            int[] column = new int[] { 0, indexes[1] };

            for (int i = 0; i < len; i++)
            {
                column[0] = i;
                object x = data.GetValue(column);
                result.SetValue(x, i);
            }

            return result;
        }
    }

    // Item
    return base.GetItemValue(indexes);
}
```

The class overrides GetIndexCount method which returns 2, because matrix element has two indexes. The method **GetSlicingImplemented** returns true, that means the variable supports slicing. **GetItemType** method returns the type of indexed data, taking into account slicing implementation. If one of indexes is less than zero it means that the data must be 'sliced' by this index. For matrix data it means that the returned data is array (matrix row or column). The **GetItemValue** method implements access to data implementing slicing analogously to **GetItemType** method.

       NOTE about indexing implementation: index type is always supposed to be integer and cannot be redefined.

Introducing function derivatives

       Extending ANALYTICS library in the sense of derivative calculation differs from the other extensions. For example, the derivative rules cannot be redefined for operators, because these rules are the base of the mathematics.

       The only derivative definition allowed is the function derivative. To define a function derivative the descendant of **FunctionalDerivative** class must be derived and its abstract methods must be implemented. The methods are:

```csharp
protected abstract string GetFunctionName(); - name of the function.
protected abstract int GetParameterCount();  - the number of function's parameters.
protected abstract int GetArgumentCount(); - the number of function's arguments.
public abstract BaseExpression Derivative(DerivativeContext context, FunctionExpression
function, string vName);
```

       The first three methods define what function the derivative rule is applied for. Note that the types of parameters and arguments are not defined. It differs from defining function for calculation because derivative calculation does not depend on the types, but calculation of functions does.

       The last method defines the rule itself. This method takes *function* parameter of **FunctionExpression** class (see class hierarchy above) and *vName* parameter – the name of variable for derivative. The *context* parameter is only needed to pass into other derivative methods. The result of the method is **BaseExpression** object. So, the method must get result expression from the input *function* expression.

In general case it is rather complicated to implement the algorithm to calculate function derivative. The methods of expression classes (see class hierarchy) must be used to build the result expression. It is because the derivative rules are **really** complicated for general case of function with many arguments and parameters. Consider the derivative of logarithm function as an example:

$$\frac{d}{dx}\left(\log_{a(x)} g(x)\right) = \frac{1}{g(x)\ln(a(x))}\frac{d}{dx}\left(g(x)\right) - \frac{\ln(g(x))}{a(x)\ln^2(a(x))}\frac{d}{dx}\left(a(x)\right)$$

These rules cannot be defined for all functions generally, they must be defined for any function individually.

But for standard elementary functions of one argument the derivative rule can be generalized, using the following mathematical formula:

$$\frac{d}{dx}\left(f(g(x))\right) = \frac{df}{dg}\frac{dg}{dx}$$

The **SimpleFunctionalDerivative** class implements this rule. This is an abstract class to implement derivative rule for functions with one argument. It overrides the base **Derivative** method (realizing the above rule). The only abstract method of the class which must be overridden in descendants is

```
protected abstract BaseExpression BaseDerivative(BaseExpression argument);
```

The *argument* parameter is g function in the formula above. The implementation of the method is rather simple for most of standard transcendental functions. Consider the code of class, implementing the derivative for sine function, as an example:

```csharp
/// <summary>
/// Sine derivative
/// </summary>
public sealed class SineDerivative : SimpleFunctionalDerivative
{
    protected override string GetFunctionName()
    {
        return "sin";
    }

    protected override BaseExpression BaseDerivative(BaseExpression argument)
    {
        return FunctionExpression.CreateSimple("cos", argument);
    }
}
```

The implementation is rather simple. The method **GetFunctionName** tells that the derivative for function with name 'sin' is defined. The method **BaseDerivative** creates new expression – function 'cos' with one given argument, because $\frac{d}{dx}(\sin(x)) = \cos(x)$.

As can be seen from the example, the methods of expression classes (see class hierarchy) can be used to easily build result expressions. There are many useful methods to create sum, difference, product, division, power or other expressions from the existing ones.

5.ANALYTICS Extensions

The ANALYTICS is a complete core library for parsing and calculating mathematical expressions. And there are some extensions of the library those implement special functionality. Extension is a library that is not needed for core algorithm but which introduces new functionality if available. Each extension can introduce the following features:
- recognition of new literal types;
- new variable types;
- operators for special types;
- functions for special argument types.

The next part contains description of existing extensions of the ANALYTICS library.

ANALYTICS Complex

The ANALYTICS Complex extension library implements operations with Complex numbers. As was said above, ANALYTICS library supports Complex literals by default. Also, ANALYTICS library supports 'explicit' operator overloading. So, some operations with complex numbers are available without any extension library. As an example, the expression '(2-I)*(3+2I)' can be evaluated correctly. But the default functionality is limited by literal expressions with operators those can be 'explicitly' overloaded.

The ANALYTICS Complex library introduces the following features:
1. Variable types: ***ComplexVariable***, ***ComplexArrayVariable***, ***ComplexMatrixVariable***, ***ComplexBlockVariable***.
2. Overloaded operators for Complex operands: '^' (power), '~' (conjugate), ''' (reciprocal).
3. Overloaded binary operators for Complex-Real operands: '+', '-', '*', '/', '^'.
4. Functions for Complex and Complex-Real arguments/parameters: ***Complex*** and ***Polar*** (create complex number by Real arguments), ***Conjugate***, ***Reciprocal***, ***Re*** (real part), ***Im*** (imaginary part), ***Arg*** (argument), ***abs*** (absolute), ***sqrt*** (square root), ***root*** (n-th root), ***pow*** (power), ***log*** (logarithm), ***ln*** (natural logarithm), ***lg*** (decimal logarithm), ***lb*** (binary logarithm), ***exp*** (exponent), trigonometric functions (***sin***, ***cos***, ***tan***), inverse trigonometric functions (***arcsin***, ***arccos***, ***arctan***), hyperbolic functions (***sinh***, ***cosh***, ***tanh***), inverse hyperbolic functions (***arcsinh***, ***arccosh***, ***arctanh***).

ANALYTICS Fractions

The ANALYTICS Fractions extension library implements analytical operations with Common (simple) fractions. The following features realized:
1. Common Fraction Literal: the library introduces support of parser to recognize fraction literals. After the literal is registered it recognizes the expression 'a/b' as fraction if 'a' and 'b' are integer numbers. NOTE: due to precedence of operators, the fraction literal must be enclosed by parentheses, if the operators outside have the higher or the same precedence as '/' operator does. For an example, '2/3+4/2' – parentheses NOT obligatory, '(2/3)*(4/2)' – parentheses ARE obligatory.
2. Variable type: ***FractionVariable***.
3. Overloaded operators for Fraction operands: '^' (power), '~' (conversion to real value), ''' (inversed fraction) and all standard binary algebraic and relational operators.
4. Overloaded standard binary algebraic operators for Real-Fraction and Complex-Fraction operands and overloaded relational operators for Real-Fraction operands.
5. Functions for Fraction arguments: **Fraction** (constructs fraction from numerator and denominator or converts double value to common fraction), ***Numerator***, ***Denominator***, ***Real*** (converts fraction to real value), ***frac*** (fractional part of a common fraction), ***int*** (integer part of a common fraction), ***sgn*** (sign of a fraction).

ANALYTICS Linear Algebra

The ANALYTICS Linear Algebra extension library introduces features to implement analytical calculations with 3D vectors and tensors and n-dimensional vectors and rectangular matrices. The following features realized[4]:
1. Variable types: ***Vector3DVariable***, ***Tensor3DVariable***.
2. Overloaded operators for ***Vector3D*** operands: '•' (dot product).
3. Overloaded operators for ***Tensor3D*** operands: ''' (transpose), '^' (power – integer, including negative values).
4. Overloaded operators for Vector-Real operands: '+', '-', '*', '/', '•'.
5. Overloaded operators for Matrix-Vector-Real operands: '+', '-', '*', '/', '^' (power – integer, including negative values).
6. Functions for ***Vector3D*** arguments/parameters: ***Vector*** (creates vector by real components), ***Length***, ***Dot***, ***Cross***, ***Angle***, ***Distance***, ***Direction***, ***Normal***, ***Area***.
7. Functions for ***Tensor3D*** arguments/parameters: ***Tensor*** (creates tensor by real components), ***Invariant1***, ***Invariant2***, ***Invariant3***, ***Transpose***, ***Inverse***, ***Symmetric***, ***Antisymmetric***, ***Invariant1***, ***Solve*** (solves linear equation system).

---

[4] ANALYTICS Linear Algebra library depends on MATHEMATICS library because it uses special types, such as Vector3D, Tensor3D and so on, from it. As these special types implement operators overloading for base mathematical operations, extension library introduces only additional operators ('explicitly' overloaded operators used automatically).

8. Functions for **Vector** (n-dimensional) arguments/parameters: **Min**, **Max**, **Length**.
9. Functions for **Matrix** (rectangular) arguments/parameters: **Min**, **Max**, **Transpose**, **Inverse**, **Minor**, **Determinant, Adjoint, Solve** (solves linear equation system).
10. Additional types: **StringVector**, **StringTensor**. These special classes implement 'analytical' vector and tensor (their components are string expressions, all operations implemented in analytical form).

ANALYTICS Mathphysics

The ANALYTICS Mathphysics extension library introduces features to work with physical entities (units of measurement and physical values)[5] in analytical expressions. The following features realized:
1. Literal types: the library supports following literals – **unit literal**, **scalar value literal**. Unit literal is sequence of symbols, representing valid notation of a unit of measurement. As there is a lot of unit names and prefixes, and the unit notation is 'rather close' to mathematical expression notation (used in ANALYTICS library) unit literals MUST BE ENCLOSED in triangle braces '‹›'[6]. The examples of valid unit literals: '**‹mm^2›**', '**‹m kg/s^2›**', '**‹m kg s^-2›**'. The scalar value literal is a real number (literal) with associated unit of measurement. In other words – Real literal + Unit literal. The real literal and the unit can be separated by space ' ' or can be NOT separated. The examples of valid scalar value literals: '**4‹m/s›**', '**2.2‹m kg s^-2›**', '**-3 ‹m/s^2›**'. NOTE: as the space symbol ' ' used in unit literals instead of product symbol (for notation being close to human) but in ANALYTICS library the symbol used for argument separator, the unit (or scalar value) literal should be enclosed in additional parenthesis when used as function parameter/argument.
2. Variable types: **UnitVariable**, **PhysicalScalarVariable**.
3. Overloaded operators for Scalar-Unit-Real operands: '**+**', '**-**', '*****', '**/**', '**^**' (power – integer, including negative values).
4. Functions for Scalar-Unit-Real arguments/parameters: **Convert** (converts values from one unit to another), **Value** (the Real value of physical scalar), **Unit** (the unit of physical scalar).

ANALYTICS Special

The ANALYTICS Special extension library introduces possibility of calculating special functions in analytical expressions. The following functions realized: **P** – Legendre's polynomial and associated Legendre's polynomial of the first kind, **J0**, **J1**, **Y0**, **Y1**, **I0**, **I1**, **K0**, **K1** – Bessel functions and modified Bessel functions of the first and second kind (of order 0 and 1).

---

[5] ANALYTICS Mathphysics library depends on PHYSICS library. For total information about units of measurement and physical values see PHYSICS manual.
[6] Do not confuse symbols of triangle braces '‹ ›' with keyboard symbols '<' – less and '>' – more.