info.lundin.math - math expression parser for .NET

by

Patrik Lundin
patrik@lundin.info
http://www.lundin.info

Copyright, License and disclaimer

Quick start – how to use the library

To use the parser in any .NET language simply copy the info.lundin.math.dll into the same directory as your source files and then reference it.

For example, using C#:

```
// Import the info.lundin.math namespace
using info.lundin.math;

// Some other imports
using System;
using System.Collections;

public class Test
{
            public static void Main( String[] args )
            {
                        // Instantiate the parser
                        ExpressionParser parser = new ExpressionParser();

                        // Create a hashtable to hold values
                        Hashtable h = new Hashtable();

                        // Add variables and values to hashtable
                        h.Add( "x", 1.ToString() );
                        h.Add( "y", 2.ToString() );

                        // Parse and write the result
                        try
                        {
                          double result = parser.Parse( "xcos(y)", h  );
                          Console.WriteLine( “Result: {0}”, result );
                        }
                        catch(ParserException ex)
                        {
                          Console.WriteLine( “Error: {0}”, ex.Message );
                        }
            }
}
```

When you compile your source you must make sure it finds the info.lundin.math.dll, with the Microsoft C# compiler this is done with the /r switch: **csc /r:info.lundin.math.dll yourfile.cs**

In Visual Studio you simply right click on the references folder in solution explorer to add a reference to the dll reference.

Don't forget to send along a copy of the info.lundin.math.dll with your program. The assembly is strongly typed so it can be added to the GAC if needed but this is generally not necessary.

If your application targets a different version of the .NET framework you may have to recompile the source code, the source code is available in the "src" directory, if you use Visual Studio you can add a reference to the project in the source directory to have it compile directly with your projects.

Short Reference

Basic operators supported:

+, -, *, /, ^, %

The **^** operator means raised to, for example x^2 is the square of x. The **%** operator is the modulo operator.

The parser supports implicit multiplication in certain cases, for example 2x, xsin(x) or (x-2y)(x+y) this can be disabled by setting the property **ImplicitMultiplication** to false, doing so will require the explicit use of the * operator or an exception is thrown. For example 2*x, x*sin(x) and (x-2*y)*(x+y)

The default is to allow implicit multiplication in expressions.

Functions supported:

sqrt, sin, cos, tan, atan, acos, asin, acotan, exp, ln, log, sinh, cosh, tanh, abs, ceil, floor, fac, sfac, round, fpart

These mostly map to the equivalent functions in System.Math except sinh, cosh, tanh, fac and sfac which are derived functions not available in System.Math. See the source code if you need to see the exact implementation of these functions.

The trigonometric functions all use radians which is also the default in System.Math, to use degrees you will have to convert the value to radians first.

The log operator is equivalent with the Math.Log10 function, using base 10. For the natural logarithm use the **ln** function. For example log(10) = 1 or ln(exp(1)) = 1, to use any other base calculate log(value) / log(base) for example log(8) / log(2) which is the binary (base 2) logarithm of 8.

All functions require parentheses around arguments or an exception is thrown, for example sin(x). Setting the property **RequireParentheses** to false disables this requirement and allows expressions such as, for example, sin5 or sinx without parentheses around the arguments 5 or x.

The order of evaluation may be confusing when not using parentheses around function arguments, for this reason it is recommended to use the default value.

Logical operators supported:

!, ==, !=, ||, &&, >, < , >=, <=

These all mean the same as their counterparts in C# but the operators all evaluate to 1.0 for true and 0.0 for false, an expression that evaluates to anything other than 1.0 is considered false by the operators. There are no bitwise operators in this parser.

Constants supported:
PI (value of System.Math.PI)
Euler (value of System.Math.E)

true (1.0)
false (0.0)
infinity (value of Double.PositiveInfinity)

Variable naming:
All variable names must start with an alphabetic letter (a-z) and can contain digits at the end but not inside the variable name.

Variable names cannot contain the name of a function or the symbol of an operator.

Examples: x, y, z, var1, var2, myverylongvariablename

Localization and decimal separators:

Make sure you understand the implications of different decimal separators in different cultures. This assembly was compiled as "culture neutral", the decimal separator may change depending on the settings on the machine it runs on.

For example if the culture is "en-US" the decimal separator is "." (1.123) while in some cultures the decimal separator is "," (1,123) which in the en-US culture is used as thousands separator.

Using the wrong decimal separator may either cause an exception or lead to unexpected results depending on the culture settings.

Nested expressions:

The parser supports nested expressions where the variable added to the hashtable itself is an expression, the expression is parsed and cached the first time the variable is encountered.

For example:

```
h.Add( "x", "cos(y)-5z" );
```

What is NOT supported however is doing circular references where the variable itself is in the expression. For performance reasons **NO attempt is made to detect circular references in this parser. Doing a circular reference WILL cause a circular loop!**

Limitations of using double:

If you use the parser for financial calculations please make sure you understand how IEEE754 floating point values work and the rounding errors that may result.

It is generally not recommended to use float or double for money applications since these are floating point values with a binary representation, for languages that have them a decimal representation is recommended, however this parser uses the default System.Math library and only operate with double values.

Adding custom operators:

The parser was not designed for this and adding custom operators is currently not supported.

If you still want to attempt this look at the comments in the source code.