# Update

## May 30, 2012

The API has been slightly changed. As of changeset d28640eceb5c instead of creating an instance of the Parser class, you should create an instance of the CompiledExpression or CompiledExpression<T> class.

The difference between the generic version and the non-generic version is that the non-generic version incurs the additional overhead of boxing the result into type Object. It helps that if you actually know what type you are expecting beforehand, you should have a slight performance increase. This is untested, but I just wanted to have a cleaner API. Otherwise most everything else on the API side remains the same.

# Introduction

This is a simple mathematical and logical C# expression evaluator using Expression trees.

C# does not include a built-in method to evaluate a string during runtime, like VBScript's Eval(). I wanted to be able to define a condition "x = c" where x was a property of an runtime object that would vary over calls, and c was a constant, and this condition should be defined in an XML file where it could be changed if needed. I also wanted to ensure that any further conditions could also be defined in this manner, without the need to write any code.

One of the options I found around the internet was to compile the code into a class, load it into memory and call the function through reflection, but I immediately rejected this solution as it was clunky and inelegant.

I then came across Pascal Ganaye's Eval3 library. The library supported passing external variables through a class - just what I needed. It performed well enough (I had to modify the existing code to accommodate the equals operator) but I felt that it was a bit difficult to maintain (had to write a function for each operator-combination type) and I was trying to move away from VB.NET.

I then started working with LINQ and Expression trees and realized I could build my own parser, create an Expression tree and compile it, and even cache the function delegate for optimum speed.

With a bit of help from Wikipedia, I borrowed an implementation of Dijkstra's Shunting-yard algorithm to handle the parsing, rewrote it in C#, threw in Expressions and had the evaluator up and running.

# Features

C# style operators

Arithmetic operators: +- \* / % ^

Relational operators: = != < > <= >=

Logical Operators: ! & | (bitwise logic) && || (short circuit logic)

Brackets ( )

Index accessor [ ]

External variables through a class

Strings: enclosed in 'single quotes' and string concatenation

DateTimes: #any valid date format#, #Now# returns DateTime.Now

true, false, null literals

Declarative typing of numbers as double and float using the d and f suffixes

Implicit conversion of numerical expressions

Member access operator (.) for any valid expression. Access properties, fields and methods of types, objects and expressions

Registry of external types and objects

Some built-in default types (bool, int, double, float, char, string, DateTime, Convert, Math)

Nested function calls (x.method(y.method(z.method()), y.method2()))

Compiled expression is cached for speed, useful if expression needs to be evaluated multiple times

Getting Started

See Usage and Sample Expressions under Documentation

Disclaimer

The parser code is in the get-it-working state and may contain bugs.

Last edited May 30 at 8:20 AM by RupertAvery, version 24

# Usage

Create an instance of CompiledExpression or CompiledExpression<T>. The parameter for the constructor is the string to parse.

To use external variables and methods, you may register types or objects using RegisterType(key, type) method. Provide an alias for the type/object as the key. Use this alias when accessing the type or object's methods or properties.

Default types are available, to use, call RegisterDefaultTypes(). This will give you bool, int, string and DateTime types to name a few, to call methods on.

Call Parse() and then Compile(). Finally call Eval() to retrieve the result. Eval will call Compile which will also call Parse if neither has been called yet - the functions are separate in order to allow benchmarking separately.

Setting the StringToParse property will update the expression and clear the compiled function, triggering another Parse/Compile the next time Eval is called.

Sample Code

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using ExpressionEvaluator;

namespace EvalTest

{

    class Program

    {

        class Vars

        {

            public DateTime myDateVar { get; set; }

            public double myDoubleVar { get; set; }

        }

        static void DoTest()

        {

            var v = new Vars() { myDoubleVar = 1234.5678 };

            string parsestr = "(vars.myDoubleVar / 4 \* 3) + (vars.myDoubleVar / (2 + 5))";

            var p = new CompiledExpression(parsestr);

            p.RegisterType("vars", v)

            p.Parse();

            p.Compile();

            Console.WriteLine("Result: {0}", p.Eval());

        }

        static void Main(string[] args)

        {

            DoTest();

            Console.ReadLine();

        }

    }

}

Sample Expressions

Here are some sample expressions to demonstrate the capabilities of the parser.

Mathematical expressions

By default, numbers without decimals are parsed as integers, and numbers with decimals are parsed as doubles. Mixing different types is allowed, and implicit conversion will be applied between expressions of different types.

"1 + 2" // returns (int) 3

 "1 + 2d + 2.5" // returns (double) 5.5

Boolean expressions

"true != false" // returns true

Strings, string concatenation

A single quote ' will have to be escaped as \' which in C# has to be escaped as \\'. A backslash will have to be escaped as \\\\

"'\\\\Foo' + 'bar' + '\\'s'" // returns \Foobar's

Registered default types

You will need to call RegisterDefaultTypes() on an instance of the class before you can access any members on the default types.

   parserInstance.RegisterDefaultTypes(); // Registers the default types

   ...

   "DateTime.Now.AddDays(5)" // returns the current time plus 5 days

   "int.Parse('42')" // returns the answer to Life, the Universe, and Everything

Registered objects, object member access

class MyClass

    {

        public double myExternalVar;

        public double getRandomNumber()

        {

            return 4; // chosen by fair dice roll

                         // guaranteed to be random

        }

    }

    ...

    var v = new MyClass();

    v.myExternalVar = 6;

    parserInstance.RegisterType("vars", v);

   "(vars.myExternalVar + 3) / 2 \* 4.5 " // returns 20.25

   "vars.getRandomNumber()"  // returns a random number

Index accessors

class MyClass

    {

         public string[] myExternalString;

         public int x;

    }

    var v = new MyClass();

    v.myExternalString = new string[] { "Hello", "there", "World!" };

    v.x = 2;

    ...

   parserInstance.RegisterType("v", v);

   "v.myExternalString[v.x/2]" // returns "there"

Member accessors

   "'Foobar'.Substring(0,3)" // returns "Foo"

   "v.myStringVar.Substring(0,x)" // returns the first x characters of the string myVarString

   "#Now#.ToString('yyyy/MM/dd hh:mm:ss')" // returns the current time formatted