# Lambda Expression Parser User Guide (Version 2.0)

| Version | Date | Remark |
|---|---|---|
| 0.1 | 08/21/2011 | Create |
| 0.2 | 08/22/2011 | • Added null-able types<br>• Added operators: new, is, typeof and sizeof.<br>• Correct operator precedence<br>• Improved parsing error message<br>• Bug fix for static method call |
| 1.0 | 8/31/2011<br><br>9/4/2011<br><br>9/12/2011 | • Fixed implicit type conversion for null value<br>• Fixed error when using namespace with assembly name<br>• Fixed error when use nested type<br>• Fixed insensitive property and field<br>• Allow empty expression to return as a null constant<br>• Fully support C# Literals<br>- Added hexadecimal integer literal: 0x123<br>- Added exponent identifier for real literal: 1e-2<br>- Added hexadecimal and Unicode escape sequence for character literals: “\xFFFF”, “\uFFFF”, “\U0000FFFF”<br>- Added verbatim string literal: @”xx\txx””xx”<br><br>See [**7. Support Literals**] for the full list of support. |
| 1.1 | 9/14/2011<br>9/19/2011<br>9/20/2011<br>9/23/2011 | • Add support to array type in inline parameter declaration: (int[,] num)=>xxxx<br>• Add support to Generic Class and Generic Method. See example in 4.6.<br>• Add support array initializer: new int[] { 1, 2, 3 }, See example in 4.7.<br>• Reduced redundant coding. |
| 2.0 | 10/19/2011 | • Add support to statements. See example in [**5. Write your statements**]<br>• Add support to new operaters, such as: x++; x--; ++x; --x; as |
|  |  |  |
|  |  |  |

## 1. Setup

Expression Parser requires **.Net Framework 4.0** or above. To use the Parser, please:

- ✓ Download library – Simpro.Expr_v2.0.zip
- ✓ Unzip Simpro.Expr_v2.0.zip to Simpro.Expr.dll
- ✓ Add reference to Simpro.Expr.dll from your .Net project

## 2. First example

```
using Simpro.Expr;

ExprParser  ep = new ExprParser();
LambdaExpression  lambda = ep.Parse("(int x, y) => 10 * (x + 1) - y");
int result = (int) ep.Run(lambda, 10, 20);      // result is 90
```

## 3. Expression Parser

You must create an instance of ExprParser before parsing any expression. One instance of ExprParser can parse multiple expressions. If you have parameters used in the expression, you must declare the parameter types before you do the parsing, and also provide the parameter values before you run the expression. You can declare parameter type and provide parameter value on the ExprParser instance and share them between all expressions parsing from this instance.
Another option is to declare the parameter type or provide parameter value inline. Inline parameter type and value will overwrite the parameter type and value defined on Parser level.
The output of expression parser is a lambda expression. You can run the expression from your code or use the helper method Run() from ExprParser object.

```
using Simpro.Expr;

ExprParser ep = new ExprParser();
ep.ParameterType["x"] = typeof(int);       // declare the parameter type on Parser level
LambdaExpression le1 = ep.Parse("x + 1");
LambdaExpression le2 = ep.Parse("(int  x) => x - 1");        // declare the parameter type in-line

ep.ParameterValue["x"] = 10;                // provide the parameter value on Parser level
int result1 = (int) ep.Run(le1);
int result2 = (int) ep.Run(le2, 20);                // provide the parameter value in-line
```

The 3rd option is to declare parameter type or provide parameter value at global level so that they can be shared between all ExprParser instances.

```
using Simpro.Expr;

ExprParser.GlobalParameterType["x"] = typeof(int);
ExprParser.GlobalParameterValue["x"] = 10;
```

## 4. Write your expression

The expression is compatible with C# expression syntax and plus a few extensions. All parameters referred in the expression must have type declared. The parameter declaration can be declared as global scope, parser scope or inline.

```
ExprParser.GlobalParameterType["x"] = typeof(int);     // global scope
ep.ParameterType["y"] = typeof(int);                   // parser scope
LambdaExpression lambda = ep.Parse("(int z) => z + 1");  // inline
```

All examples of this section are supposed you have referred Simpro.Expr namespace and created ExprParser instance already as of below.

```
using Simpro.Expr;

ExprParser ep = new ExprParser();
```

### 4.1 Expression without parameter

```
LambdaExpression lambda;

// math expression
lambda = ep.Parse("1 * (2 + 3)");
int result = (int) ep.Run(lambda);                    // result is 5

// logical expression
lambda = ep.Parse("(5+1 > 6) && (5 – 1 <= 4)");
bool result = (bool) ep.Run(lambda);                  // result is false

// conditional expression
lambda = ep.Parse("(10 * 15 – 11) > 100 ? 'right result' : 'wrong result'");      // string constant is enclosed by either ' or \"
int result = (int) ep.Run(lambda);                    // result "right result"
```

### 4.2 Expression with parameter

Inline parameter type:

```
LambdaExpression lambda;

// inline parameter type and value
lambda = ep.Parse("(int x, y) => (x + 1) * y");
int result = (int) ep.Run(lambda, 10, 2);                  // result is 22
```

Parser scope parameter type and value:

```
LambdaExpression lambda;

// parser scope parameter type and value. declare once and be reused by current parser instance
ep.ParameterType["x"] = typeof(int);
ep.ParameterValue["x"]=  10;

lambda = ep.Parse("x + 1");                                // parse first expression with 'x'
int result = (int) ep.Run(lambda);                         // result is 11

lambda = ep.Parse("x - 1");                                // parse another expression with 'x'
result = (int) ep.Run(lambda);                             // result is 19
```

Global scope parameter type and value

```
LambdaExpression lambda;

// global scope parameter type and value. declare once and be reused by all parser instances
ExprParser.GlobalParameterType["x"] = typeof(int);
ExprParser.GlobalParameterValue["x"]=  10;

ExprParser  ep1 = new ExprParser();                        // first parser
lambda = ep1.Parse("x + 1");                               // parse first expression with 'x'
int result = (int) ep1.Run(lambda);                        // result is 11

ExprParser  ep2 = new ExprParser();                        // first parser
lambda = ep2.Parse("x - 1");                               // parse another expression with 'x'
result = (int) ep2.Run(lambda);                            // result is 19
```

You can declare parameter type as global scope and provide parameter value at Parser scope.

## 4.3 Use ASP.Net system data type

Build in C# types and constant values can be used in expression directly.

| Build in C# types | .Net Framework Type | Constant |
|---|---|---|
| bool | System.Boolean | true, false |
| char | System.Char | |
| string | System.String | null |
| byte | System.Byte | |
| sbyte | System.SByte | |
| short | System.Int16 | |
| ushort | SystemUInt16 | |
| int | System.Int32 | 9 |
| uint | System.Uint32 | 9U, 9u |
| long | System.Int64 | 9L, 9l |
| ulong | System.Uint64 | 9UL, 9ul |
| float | System.Single | 9.0F, 9.0f |
| double | System.Double | 9.0D, 9.0d |
| decimal | System.Decimal | 9.0M, 9.0m |
| enum | System.Enum | |
| object | System.Object | |

```
LambdaExpression lambda;

// using build-in C# data type and constant
lambda = ep.Parse("(int x; double y) => (x + 1.0f) * y + 10.0d");       // build in type int and double. Constant 1.0f and 10.0d
double result = (double) ep.Run(lambda, 10,  2.0D);                    // result is 32.0d
```

You can also use other ASP.Net data types by adding namespace as below. Namespace can only add in global scope.

```
LambdaExpression lambda;

// add namespace
ExprParser.Using.Add("System.Collection");                             // add namespace for default assembly mscorlib
ExprParser.Using.Add("System.Linq.Expressions", "System.Core");        // add namespace with assembly if its other than mscorlib

lambda = ep.Parse("(ArrayList  x) => Expression.Constant(x.Length)");  // use static method and ArrayList property
```

Null-able data types are also supported.

| Build in C# types | .Net Framework Type | Constant |
|---|---|---|
| bool? | System.Nullable<Boolean> | |
| char? | System.Nullable<Char> | |
| byte? | System.Nullable<Byte> | |
| sbyte? | System.Nullable<Sbyte> | |
| short? | System.Nullable<Int16> | |
| ushort? | System.Nullable<Uint16> | |
| int? | System.Nullable<Int32> | |
| uint? | System.Nullable<Uint32> | |
| long? | System.Nullable<Int64> | |
| ulong? | System.Nullable<Uint64> | |
| float? | System.Nullable<Single> | |
| double? | System.Nullable<Double> | |
| decimal? | System.Nullable<Decimal> | |

```
LambdaExpression lambda;

// using build-in C# Nullable data type
lambda = ep.Parse("(int? x) => (x==null)? 0 : x + 1");          // build in Nullable type int?
double result = (double) ep.Run(lambda, 10);                    // result is 11
```

## 4.4 Use user data type

To use user defined data type, you need to add namespace for the data type first as below. If the namespace is from the assembly that calling ExprParser.Parse(), you only need to add namespace to Using list. Else you also need to specify the assembly name.

```
LambdaExpression lambda;

namespace MyNamespace
{
        public class MyClass
        {
                public int x = 1;
                public int AddOne() { return x + 1; }
        }
}

ExprParser.Using.Add("MyNamespace");                          // add namespace for ExprParser.Parse() caller assembly
ExprParser.Using.Add("MyOtherNamespace",  "MyOtherAssembly"); // provide assembly name if other than caller assembly
// refer user defined class
lambda = ep.Parse("(MyClass  obj) => obj.x + obj.AddOne()");  // use user defined field and method
```

## 4.5 Type Cast

Use (T) x to cast one data type to another.

```
LambdaExpression lambda;

// using build-in C# data type and constant
lambda = ep.Parse("(double x; decimal y) => ((decimal) x > y)? 'x is greater than y.' : 'x is less than y.'");     // double cast to decimal
double result = (double) ep.Run(lambda, 10,  2.0D);                    // result is 32.0d
```

## 4.6 Generic class and method

```
LambdaExpression lambda;

// example generic class with static and instance generic method
public class MyGenericClass<T>
{
    public static int CheckStaticGeneric<T1, T2>(T1 x, T2 y)
    {
        if (x is int || y is int) return 1; else return 0;
    }
    public int CheckInstanceGeneric<T1, T2>(T1 x, T2 y)
    {
        if (x is int || y is int) return 1; else return 0;
    }
}

// using static generic method
lambda = ep.Parse(@"MyGenericClass<int>
      .CheckStaticGeneric<long,System.Collections.Generic.Dictionary<string, long>>
      (10L,new System.Collections.Generic.Dictionary<string,long>())");
int res = (int)ep.Run(lambda);

// using static generic method
lambda = ep.Parse(@"new MyGenericClass<int>()
      .CheckInstanceGeneric<long,System.Collections.Generic.Dictionary<string, long>>
      (10L,new System.Collections.Generic.Dictionary<string,long>())");
int res = (int)ep.Run(lambda);
```

## 4.7 Array and array initializer

```
LambdaExpression lambda;

lambda = ep.Parse("new int[] {1,2,3}.Length"); int res = (int)ep.Run(lambda); // by initializer
lambda = ep.Parse("new int[3,4].Length"); int res = (int)ep.Run(lambda);// by dimension
```

Multiple dimension array is only supported by using dimension, not supported by using initializer. Below parsing return error.

```
lambda = ep.Parse("new int[] {{1,2},{3,4}}.Length"); // throw exception
```

## 5. Write your statements

From version 2.0, ExprParser also supports C# compatible statements. The expression can contains multiple statements. All statements parsed as a unit is like a C# function that can have a return value. Below is a simple statement parsing example.

```
const string statement_sample = @"
    int x = 10;
    int y = (x + 1) * x;
    return y;
";
ExprParser ep = new ExprParser();
LambdaExpression lambda = ep.Parse(statement_sample);
int res = (int) ep.Run(lambda);

// parsing and running above statements block is equivalent to run below function

public int test_statement_sample()
{
    int x = 10;
    int y = (x + 1) * x;
    return y;
}
Debug.Assert(res == test_statement_sample());
```

### 5.1 Statement block and local variable

Statement block is defined by "{" and "}". Statement block can have multiple level. Every block can declare its own local variables.

```
const string statement_sample = @"(int x)=>
    if(x>0)
    {
       int y = 10;
       return x * y;
    }
    else
    {
       int y = -10;
       return x * y;
    }
";

ExprParser ep = new ExprParser();
LambdaExpression lambda = ep.Parse(statement_sample);
int res = (int) ep.Run(lambda, 1);
```

## 5.2 Return statement

From any place of the statement list, `return` can be used to return a final result value. Not like C# function, all return values in the expression haven't to be the same type. The Parser will anyway return a `object` type value. If there is no `return` statement in the statement list, the result value of the whole statement list is defined by the last statement value. If the last statement cannot be determined, the return value will be `void`.

```
const string statement_sample1 = @" (bool returnText)=>
    if (returnText)
        return ""1"";
    else
        return 1;    // return value can be different type
";
ExprParser ep = new ExprParser();
LambdaExpression lambda = ep.Parse(statement_sample);
Debug.Assert("1" == (string)ep.Run(lambda, true));
Debug.Assert(1 == (int)ep.Run(lambda, false));

const string statement_sample2 = @"
    int x = 10;
    int y = x++;
    y = x * y;       // return value will be the last statement: x * y
";
const string statement_sample2 = @"
    int x = 10;
    if(x>0)
       x++;
    else
       x--;   // return value will be void because the last statement cannot be determined
";
```

## 5.3 If statement

**If** statement can be used with keywords: `if`, `else if`, `else`.

```
const string statement_sample = @"(int branch)=>
    if (branch == 1)
        return ""if"";      // control a single statement
    else if (branch == 2)
    {
        return ""else if""; // control a statement block
    }
    else
        return ""else"";
";
```

## 5.4 While statement

**While** statement can be used with keywords: `while`, `break`, and `continue`.

```
const string statement_sample = @"(int x)=>
    int y = 0;
    while (x > 0)
    {
        y += x--;
        if(y > 100) break;
    }
    return y;
";
```

## 5.5 Do-while statement

**Do-while** statement can be used with keywords: `do`, `while`, `break`, and `continue`.

```
const string statement_sample = @"(int x)=>
    int x = 10, y = 0;
    do
    {
        y += x--;
        if (y > 100) break;
    } while (x > 0);
    return y;
";
```

## 5.6 For statement

**For** statement can be used with keywords: `for`, `break`, and `continue`.

```
const string statement_sample = @"
    int y = 0;
    for (int x = 0; x < 10; x++)
    {
        y += x;
        if (y > 100) break;
    }
    return y;
";
```

## 5.7 Foreach statement

**Foreach** statement can be used with keywords: `foreach`, `break`, and `continue`.

```
const string statement_sample = @"
    int[] x = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int y = 0;
    foreach (int n in x) y += n;
    return y;
";
```

## 5.8 Switch-case statement

**Switch-case** statement can be used with keywords: `switch`, `case`, `default`, and `break`.

```
const string statement_sample = @"(int sw_value)=>
    int res = 0;
    switch (sw_value)
    {
        case 1:
            res = 1; break;
        case 2:
        case 3:
            res = 3; break;
        default:
            res = 0; break;
    }
    return res;
";
```

## 5.9 Try-catch statement

**Try-catch** statement can be used with keywords: `try`, `catch`, `finally`, and `throw`.

As in C#, `try`, `catch`, `finally` can only go with statement block. `Single statement is not allowed.`

```
const string statement_sample = @"(int input_val)=>
    int res = 0;
    try
    {
        if (input_val == 0)
             throw new System.DivideByZeroException();
        else
             res = 100 / input_val;
    }
    catch (DivideByZeroException ex) { Console.WriteLine(""Attempted divide by zero.""); }
    catch { throw; }
    finally { res = res + 1; }
    return res;
";
```

## 6. Support operators

| operator | description | example |
|---|---|---|
| + | Add, plus | 10 +3, +10 + 3 |
| ++ (2.0) | Increment | x++, ++x |
| -- | Decrement | x--, --x |
| - | Subtract, negate | 10 – 3, -10 + 3 |
| * | Multiply | 10 * 3 |
| / | Divide | 10 / 3 |
| % | Modulo | 10 % 3 |
| ** | Power (extension operator) | 10 ** 3 |
| & | Bit and | 10 & 3 |
| \| | Bit or | 10 \| 3 |
| ^ | Bit exclusive or | 10 ^ 3 |
| >> | Bit right shift | 10 >> 3 |
| << | Bit left shift | 10 << 3 |
| && | Logical and | x>y && x>0 |
| \|\| | Logical or | x>y \|\| x>0 |
| > | Greater than | x > y |
| >= | Greater than or equal | x >= y |
| < | Less than | x < y |
| <= | Less than or equal | x <= y |
| == | Equal | x == y |
| ?? | Coalesce | x ?? y |
| ? : | Conditional | (x==y)? true : false |
| [ ] | Array index | x[i] |
| is | Type is | o is int |
| as (2.0) | Type as | v as Type |
| new | New | new MyClass() |
| typeof | Type of | typeof(MyClass) |
| sizeof | Size of | sizeof(int) |
| = (2.0) | Assignment | y = x |
| += (2.0) | Addition assignment | y += x |
| -= (2.0) | Subtraction assignment | y -= x |
| *= (2.0) | Multiplication assignment | y *= x |
| /= (2.0) | Division assignment | y /= x |
| %= (2.0) | Modulus assignment | y %= x |
| **= (2.0) | Power assignment (extension operator) | y **= x |
| &= (2.0) | Bitwise and assignment | y &= x |
| \|= (2.0) | Bitwise or assignment | y \|= x |
| ^= (2.0) | Exclusive or assignment | y ^= x |
| <<= (2.0) | Left shift assignment | Y <<= x |
| >>= (2.0) | Right shift assignment | Y >>= x |

(2.0) supported from version 2.0.

## 7. Support Literals

| Category | Literals | example |
|---|---|---|
| Boolean | boolean | true, false |
| Interger | uint suffix (u, U) | 12u, 100U |
|  | long suffix (l L) | 12L |
|  | ulong suffix (UL Ul uL ul LU Lu lU lu) | 12UL |
|  | hexadecimal integer literal (0x, 0X) | 0xFF |
| Real | float suffix (f, F) | 10.01f |
|  | double suffix (d, D) | 10.01d |
|  | decimal suffix (m, M) | 10.01m |
|  | exponent (e, E) | 11e-2 |
| Character | simple escape sequence | \’ \” \0 \a \b \f \n \r \t \v |
|  | hexadecimal escape sequence (\x) | \xF \xFF \xFFF \xFFFF |
|  | unicode character escape sequence (\u \U) | \uFFFF \U0000FFFF |
| String | verbatim string (@) | @”xx””xx”  @’xxx’’xxx’ |
| Null | null | null |
|  |  |  |

## 8. Support statements

| Category | Keywords |
|---|---|
| Statement block | {,  } |
| Local varible | <all data types> |
| Assignment | =, +=, -=, *=, /=, %=, **=, &=, \|=, ^=, <<=, >>= |
| Return | return |
| If | if – else if - else |
| Switch | switch – case – default - break |
| Do | do – while – break – continue |
| While | while – break – continue |
| For | for – break – continue |
| Foreach | foreach – in – break – continue |
| Exception | try – catch – finally – throw |