Introduction

This package provides the API for parsing and expression execution written in C#. It is specially designed to work with the Unity on various platforms. Since it is written in C# 3.5, it should work with any version of Unity.

It is tested to work on: * IOS * Android * WebGL * PC/Mac

It should work on any other platforms.

API * CSharpExpression * Evaluate * Parse * AotCompilation * RegisterFunc * RegisterForFastCall

Example

Parsing C# expression into **System.Linq.Expression.Expression[T]**:

```
var mathExpr = "Math.Max(x, y)";
var exprTree = CSharpExpression.Parse<double, double, double>(mathExpr, arg1Name:
"x", arg2Name: "y")
// exprTree -> Expression<Func<double, double, double>>
```

Evaluating C# expression:

```
var arifExpr = "2 * (2 + 3) << 1 + 1 & 7 | 25 ^ 10";
var result = CSharpExpression.Evaluate<int>(arifExpr);
// result -> 19
```

Parser

The parser recognizes the C# 4 grammar only. It includes:

- Arithmetic operations
- Bitwise operations
- Logical operations
- Conditional operator
- Null-coalescing operator
- Method/Delegate/Constructor call
- Property/Field access
- Indexers
- Casting and Conversion
- Is Operator
- As Operator
- TypeOf Operator
- Default Operator

- Expression grouping with parentheses
- Checked/Unchecked scopes
- Aliases for Built-In Types
- Null-conditional Operators
- Power operator `**`
- Lambda expressions
- "true", "false", "null"

Nullable types are supported. Generics are supported. Enumerations are supported. Type inference is not available and your should always specify generic parameters on types and methods.

Known Types

For security reasons the parser does not provide access to any types except: * argument types * primitive types * Math, Array, Func<> (up to 4 arguments) types

To access other types your should pass **typeResolver** parameter in **Parse** and **Evaluate** method:

```
var typeResolver = new KnownTypeResolver(typeof(Mathf), typeof(Time));
CSharpExpression.Evaluate<int>("Mathf.Clamp(Time.time, 1.0f, 3.0f)", typeResolver);
```

If you want to access all types in **UnityEngine** you can pass **AssemblyTypeResolver.UnityEngine** as typeResolver parameter.

For security reasons any member invocation on **System.Type** will throw exceptions until **System.Type** is added as known type.

AOT Execution

You can compile and evaluate expression created by **System.Linq.Expression** and execute it in AOT environment where it is usually impossible.

```
var expr = (Expression<Func<Vector3>>)(() => new Vector3(1.0f, 1.0f, 1.0f));
var fn = expr.CompileAot();

fn; // -> Func<Vector3>
fn(); // -> Vector3(1.0f, 1.0f, 1.0f)
```

iOS, WebGL and most console platforms use AOT compilation which imposes following restrictions on the dynamic code execution:

- only **Expression<Func<...>>** could be used with **CompileAot()** and Lambda types
- only static methods using primitives (int, float, string, object ...) are optimized for fast calls
- all used classes/methods/properties should be visible to Unity's static code analyser

See Also * AOT Exception Patterns and Hacks * Ahead of Time Compilation (AOT)

WebGL and iOS

- Only Func<> (up to 4 arguments) Lambdas are supported
- Instance methods invocation performs slowly due reflection
- Moderate boxing for value types (see roadmap)

You can ensure that your generic Func<> pass AOT compilation by registering it with **AotCompilation.RegisterFunc**

```
AotCompilation.RegisterFunc<int, bool>(); // template: RegisterFunc<Arg1T, ResultT>
```

Improving Performance

You can improve the performance of methods invocation by registering their signatures in **AotCompilation.RegisterForFastCall()**.

```
// Supports up to 3 arguments.
// First generic argument is your class type.
// Last generic argument is return type.

AotCompilation.RegisterForFastCall<InstanceT, ResultT>()
AotCompilation.RegisterForFastCall<InstanceT, Arg1T, ResultT>()
AotCompilation.RegisterForFastCall<InstanceT, Arg1T, Arg2T, ResultT>()
AotCompilation.RegisterForFastCall<InstanceT, Arg1T, Arg2T, Arg3T, ResultT>()
```

Example:

```
public class MyVectorMath
{
    public Vector4 Dot(Vector4 vector, Vector4 vector);
    public Vector4 Cross(Vector4 vector, Vector4 vector);
    public Vector4 Scale(Vector4 vector, float scale);
}

// register Dot and Cross method signatures
AotCompilation.RegisterForFastCall<MyVectorMath, Vector4, Vector4, Vector4>();
// register Scale method signature
AotCompilation.RegisterForFastCall<MyVectorMath, Vector4, float, Vector4>();
```

Roadmap

You can send suggestions at support@gamedevware.com

- Expression serialization
- Void expressions (`System.Action` delegates)
- Parser: Delegate construction from method reference
- Parser: Type inference for generics

- Parser: Full C#6 syntax
- Parser: Extension methods
- Parser: Type initializers, List initializers
- Custom editor with auto-completion for Unity

Contacts

Please send any questions at support@gamedevware.com

License