Expressions Evaluation Engine (E3), Technical Spec

Purpose and Scope

Expressions evaluation engine (E3) can support querying, filtering and some other use cases for expressions evaluation:

- Ad-hoc conditions, applied on arbitrary objects
    - Applications: precomputed filters, object stream processing, rule-based ACLs
    - Objects: domain data, wire data contracts, arbitrary values
- SQL-like query processors
    - Applications: Where, Select, Update … Set, Order By, Group By, aggregates
    - Fields, input arguments, arbitrary constants and non-deterministic functions

SQL-like query processors can extend and use E3 engine at run-time to support advanced features such as various flavors of IN operator, expressions in the select statement etc. Single query may make use of multiple individual expressions, combining and evaluating them according to appropriate execution plan. E3 itself will only provide a generic execution engine, with support for:

- Arguments of arbitrary types
- Custom N-ary functions and atoms
- Custom extensions for the analyzer (allowing injection of custom ExpressionTrees)
- Complete runtime configuration, no need to re-compile the engine code when adding custom extensions

First goal for E3 is to deliver maximum speed for repetitive evaluations, e.g. where the same expression is evaluated over hundreds of millions of input values. Hence the three implementation requirements:

- Emit binary code at runtime
- Emit thread-safe stateless code
- Avoid dynamic memory allocation inside the evaluator, unless explicitly required by expression

Second goal for E3 is smooth integration with C# code, data structures and garbage collector. Thus, implementation will be built in C#, using Expression Trees and garbage-collectible lambdas (Funcs and Actions), and compilation APIs must support heavily concurrent clients.

Engine's input is its runtime configuration (factory object) and a text string for parsing, analysis and compilation.

Outputs are a parse tree (AST – abstract syntax tree), Expression tree, and a compiled .NET lambda.

Language & APIs

E3 is distributed as a .NET 4.5 assembly DLL, paired with 3-rd party parsing library (custom build of Irony.dll, source from http://irony.codeplex.com/), debug symbols and XML comments files.

E3's application programming interface exposes:

- Parsing library dependency: `ParseTree`, `ParseTreeNode`
    - Objects of these types support parser's and compiler's extensibility features
- Runtime factory object: `interface IExpressionEvaluatorRuntime`
    - Implemented by public type `ExpressionEvaluatorRuntime`
    - Registration of custom atoms, either as pre-compiled lambdas or as `Expression` generators (injection of Expression Trees)
    - Parsing an expression text, produces `ParseTree`
    - Analyzing `ParseTree`, produces `LambdaExpression`
    - Compiling `LambdaExpression` into `Func` or `Action`
- Ad-hoc functor signature convention for general-purpose expressions
    - `Func<…>, Action<…>,` with zero or more arguments
- Recommended functor signature convention for certain kinds of extensions
    - `Func<TContext, T>,` where `TContext` is any type
    - Predefined name for the context argument, "@Context"
    - Apply when the number of arguments passed into expression may be very large (but not all of them are used at the same time) to reduce the overhead
    - Also helps to improve maintainability of client code

Available Data Types

Usage Rules

In scope of expression engine, data types are driven by .NET framework's native value types and reference types. In external application such as SQL-like query processor the data type system will be built around `System.Data.DbType` enumeration, and its values are mapped onto .NET native data types.

When casting or converting values, you supply desired data type's name in single quotes. These string names are automatically prepended with "System." and mapped onto .NET CLR native types. If the data type you reference by name in quotes is not part of System namespace, you will get a compile-time error.

Expression Engine itself does not require arguments to be of certain types unless you ask it to execute some operation. In the context of some operator or function, expression may try to cast your argument to required type and may fail either at compile time or at run time.

Many data types are not supported by any E3 operator or function. Nevertheless, you can reference atoms that return values of those types. Type restrictions are only applied when try to do something with the value other than just passing it around (e.g. when you apply cast, convert, arithmetic etc.).

String comparisons are ordinal, case-insensitive, culture-agnostic. Beware, “culture-agnostic” is different than “culture-invariant”: we use .NET’s native `StringComparer.OrdinalIgnoreCase` to maximize performance, and this may produce undesirable results in certain special situations with Unicode characters, but is sufficient for majority of cases.

8-bit

Byte, SByte, Boolean

16-bit

Int16, UInt16, Char

32-bit

Int32, UInt32, Single

64-bit

Int64, UInt64, Double, DateTime, TimeSpan

128-bit

Decimal, Guid

Reference

Object, String

Reference List of Predefined Operators and Functions

Unary Operators

-	Negates operand	Numeric
+	Decorates positive numeric value	Numeric
~	Bitwise complement	Integer
NOT	Boolean negation	Boolean
IS NULL	Test for null value (not same as test for empty!)	Boolean
IS NOT NULL	Test for not-null value (not same as test for non-empty!)	Boolean

Binary Operators

+, -, *, /, %	Arithmetics	Numeric and Numeric

+	Non-numeric addition or concatenation	String and String DateTime and TimeSpan TimeSpan and TimeSpan TimeSpan and DateTime
-	Non-numeric subtraction	DateTime and DateTime DateTime and TimeSpan
=, >=, <=, <>, != , !>, !<	Comparison, yields Boolean	String and String (case-insensitive) Numeric and Numeric DateTime and DateTime
AND, OR, XOR	Boolean	Boolean and Boolean
&, |, ^	Bitwise arithmetics	Numeric and Numeric

Other Operators

CASE [arg] **WHEN** [val1] **THEN** [expr1] **ELSE** [expr2] **END**	Analog of switch statement in C#	Data types of expr1…exprN must match data type of arg
CASE WHEN [cond1] **THEN** [expr2] **ELSE** [expr3] **END**	Analog of series of if-then-else statements in C#	Data types of cond1…condN must be Boolean Data types of expr1…exprN must match
[arg] **IN** ([set]), [arg] **NOT IN** ([set])	Returns Boolean true or false, depending on whether argument equals one of the set	[set] must be an explicit comma-separated list of constant literals of numeric or string type
[arg] **BETWEEN** [left] **AND** [right], [arg] **NOT BETWEEN** [left] **AND** [right]	Translates to either [arg] >= [left] AND [arg] <= [right], or [arg] < [left] OR [arg] > [right]	Data types of [arg], [left] and [right] must match

Functions & Literals

NULL	Untyped NULL literal
True, False	Predefined Boolean constant literals
IsNull ([arg])	Same as "arg IS NULL"
IfNull ([arg], [def])	If arg is null, returns def, otherwise returns arg
IsDefault ([arg])	Returns true if argument equals default value for its data type (zero for numeric or Guid, false for Boolean, nil pointer for string and binary, zero date for datetime)
Default ('[type]')	Returns default value of supplied data type (zero for numeric or Guid, false for Boolean, nil pointer for string and binary, zero date for datetime)
IsNaN ([arg])	Returns true if argument is not a number (only for Single and Double floating point values)
IsInfinity ([arg])	Returns true if argument is a positive or negative infinity (only for Single and Double floating point values)
PositiveInfinity ([arg])	Returns positive infinity value (Double floating point value)
NegativeInfinity ([arg])	Returns negative infinity value (Double floating point value)

Convert ([arg], '[type]')	Attempts to convert value of one data type into value of another data type. Arguments can be of any type, will throw run-time error if types are incompatible.
Cast ([arg], '[type]')	Attempts to cast value of one data type into value of another data type. Arguments must be of compatible types. At this time, use it to cast between numeric types only.
StartsWith ([str], [prefix])	Returns true if first argument begins with second argument. Arguments must be strings. Case-insensitive.
EndsWith ([str], [suffix])	Returns true if first argument ends with second argument. Arguments must be strings. Case-insensitive.
Contains ([str], [infix])	Returns true if first argument contains, begins with or ends with second argument. Arguments must be strings. Case-insensitive.
ToDateTime ([str], [fmt]) **ToDateTime** ([year], [month], [day], [hour], [minute], [second]) **ToDateTime**([int64])	First version parses first string argument into datetime using format string (second argument). Second version produces a datetime out of six integers . Third version produces a datetime out of Int64 value previously generated by DateTime.ToBinary() – in order to preserve UTC ticks/kind marker.

Handling NULL

Kinds of NULL

Null values come in different flavors. First, there is a "NULL" literal, so that you can write UPDATE statement that sets some field to NULL, or maybe some expression that returns NULL. Also, a NULL value may exist implicitly and E3 lets you test any expression for being NULL.

Technically, NULL literal is an untyped value. E3 compiler automatically translates NULL literals into the default empty constant value for appropriate type, and the type is usually derived from context. Internally, NULL literal is passed around as a special type called "`VoidMarker`" – because NULL literal is indeed untyped and void.

NULL values are represented in different ways for reference and value types. For reference types, NULL value is passed around as a nil pointer. For value types, there is a utility generic structure called `UnboxableNullable<T>` that can be instantiated with any value type. When you use NULL literal, compiler will instantiate an `UnboxableNullable<VoidMarker>` and pass it around as-is until it is able to derive appropriate type to cast to: for reference type it will get transformed into "null" constant, for value type it will be transformed into default(`UnboxableNullable<T>`).

There is a reason not to use .NET's standard `Nullable<T>` structure to pass nullable-type values. When boxed into object, standard nullables are represented as a nil pointer and valuable state information is lost. This particular boxing issue is a problem for E3 compiler, because it needs to manipulate `ConstantExpression` objects holding null nullable values without loss of state. E3 compiler also never uses .NET boxing, so there is no performance impact caused by use of special nullable generic.

Semantics of using NULL

E3 expression	Return type
`NULL`	`UnboxableNullable<VoidMarker>`
Binary and unary operators with `NULL` literal or nullable types	Always non-nullable value or reference type, e.g. `UnboxableNullable` is stripped off, nullable-type values are replaced with default values of appropriate type
`CASE` with `NULL` literal or nullable types	`UnboxableNullable<T>`, e.g. compiler replaces constant of type with proper value of appropriate type – as soon as it knows what type to use
Function calls with `NULL` literal or nullable types	Defined by function's signature

Use Cases

Arithmetic and Boolean Operations

```
Assert.AreEqual(1.5*2.6, ((Func<Double>)m_runtime.Compile(
"1.5 * 2.6", typeof(Double)))());
Assert.AreEqual(-1, ((Func<int>)m_runtime.Compile(
"1 + 2*(3-4)", typeof(Int32)))());
Assert.IsTrue(((Func<bool>)m_runtime.Compile(
"1 > -1.5", typeof(bool)))());
Assert.IsTrue(((Func<bool>)m_runtime.Compile(
"'xyz' > 'abc'", typeof(bool)))());
Assert.AreEqual(Double.NegativeInfinity, ((Func<Double>)m_runtime.Compile(
"-1.0/0", typeof(Double)))());
Assert.IsTrue(((Func<bool>)m_runtime.Compile(
"false OR true", typeof(bool)))());
Assert.IsTrue(((Func<bool>)m_runtime.Compile(
"false XOR true", typeof(bool)))());
Assert.IsTrue(((Func<bool>)m_runtime.Compile(
"NOT false", typeof(bool)))());
```

Arguments

```
var eval = (Func<string, string, bool>) m_runtime.Compile(
          "EndsWith(@arg1, @arg2)", typeof (bool),
          new Tuple<string, Type>("@arg1", typeof (string)),
          new Tuple<string, Type>("@arg2", typeof(string)));

Assert.IsTrue(eval("abcde", "de"));
Assert.IsFalse(eval("abcde", "ee"));
```

Type Casting

```
var eval4 = (Func<Single, bool>)m_runtime.Compile(
"@arg1 = cast(1 + 0.5, 'Single')",
typeof(Boolean),
new Tuple<string, Type>("@arg1", typeof(Single)));

Assert.IsTrue (eval4(1.5f));
Assert.IsFalse(eval4(1.4f));

eval4 = (Func<Single, bool>)m_runtime.Compile(
"@arg1 = cast(1 + 0.5, 'int32')",
typeof(Boolean),
new Tuple<string, Type>("@arg1", typeof(Single)));

Assert.IsFalse(eval4(1.5f));
Assert.IsFalse(eval4(1.4f));
Assert.IsTrue (eval4(1.0f));
```

Type Conversion

```
var eval6 = (Func<int, string>) m_runtime.Compile(
"convert(@var*2, 'string') + 'xyz'",
typeof (string), new Tuple<string, Type>("@var", typeof (int)));

Assert.AreEqual("10xyz", eval6(5));

var eval5 = (Func<string, int>) m_runtime.Compile(
"cast(convert(@var, 'Double') * 2, 'int32')",
typeof(int), new Tuple<string, Type>("@var", typeof(string)));

Assert.AreEqual(8, eval5("4"));
Assert.AreEqual(8, eval5("4.1"));
```

Case Statement

```
var eval = (Func<Int32>) m_runtime.Compile(
"case when 1=2 then 3 when 2=3 then 5 else 4 end", typeof(Int32));
Assert.AreEqual(4, eval());

var eval3 = (Func<bool>) m_runtime.Compile("
case true when false then true else true end", typeof(Boolean));
Assert.IsTrue(eval3());
```

DateTime and TimeSpan

```
var december132013 = new DateTime(2013, 12, 13);
var addOneDay = TimeSpan.FromDays(1);
var addOneHour = TimeSpan.FromHours(1);
var subtractOneDay = TimeSpan.FromDays(-1);

Assert.AreEqual(m_runtime.Compile<string, DateTime>(
    "convert(@context, 'DateTime')")("12-13-2013"), december132013);
Assert.AreEqual(m_runtime.Compile<string, TimeSpan>(
    "convert(@context, 'TimeSpan')")("1.00:00:00"), addOneDay);
Assert.AreEqual(m_runtime.Compile<string, TimeSpan>(
    "convert(@context, 'TimeSpan')")("-1.00:00:00"), subtractOneDay);

var eval = m_runtime.Compile<DateTime, bool>(
    "@context <= convert('12-13-2013', 'DateTime')");
Assert.IsTrue(eval(december132013));
Assert.IsTrue(eval(december132013.AddDays(-1)));
Assert.IsFalse(eval(december132013.AddHours(1)));

eval = m_runtime.Compile<DateTime, bool>(
    "@context <= convert('13-Dec-2013', 'DateTime')");
Assert.IsTrue(eval(december132013));
Assert.IsTrue(eval(december132013.AddDays(-1)));
Assert.IsFalse(eval(december132013.AddHours(1)));
```

```
var eval2 = m_runtime.Compile<DateTime, TimeSpan>(
    "@context - convert('13-Dec-2013', 'DateTime')");
Assert.AreEqual(addOneDay, eval2(december132013.AddDays(1)));

var eval3 = m_runtime.Compile<DateTime, DateTime>(
    "@context + convert('01:00:00', 'TimeSpan')");
Assert.AreEqual(december132013 + addOneHour, eval3(december132013));
```

Custom Functions, Extending Runtime

```
Func<string, string, bool> customEndsWith =
(s1, s2) => s1 != null && s2 != null && s1.EndsWith(
                  s2, StringComparison.OrdinalIgnoreCase);

m_runtime.RegisterAtom(
      new AtomMetadata("CustomEndsWith", (object)customEndsWith));

var eval = (Func<string, string, bool>) m_runtime.Compile(
         "CustomEndsWith(@arg1, @arg2)", typeof (bool),
         new Tuple<string, Type>("@arg1", typeof (string)),
         new Tuple<string, Type>("@arg2", typeof(string)));

Assert.IsTrue(eval("abcde", "de"));
Assert.IsFalse(eval("abcde", "ee"));
```

Custom Functions, Extending Analyzer

```
m_runtime.RegisterAtom(new AtomMetadata("CustomEndsWith", (root, state) =>
{
      Expression value;
      Expression pattern;
      var method = PrepareStringInstanceMethodCall("EndsWith", root, state,
out value, out pattern);

      return Expression.Condition(
      Expression.ReferenceEqual(Expression.Constant(null), value),
      Expression.Constant(false),
      Expression.Condition(
            Expression.ReferenceEqual(Expression.Constant(null), pattern),
            Expression.Constant(false),
            Expression.Call(value, method, pattern,
            Expression.Constant(StringComparison.OrdinalIgnoreCase))));
}));

var eval = (Func<string, string, bool>) m_runtime.Compile(
      "CustomEndsWith(@arg1, @arg2)", typeof (bool),
      new Tuple<string, Type>("@arg1", typeof (string)),
      new Tuple<string, Type>("@arg2", typeof(string)));

Assert.IsTrue(eval("abcde", "de"));
Assert.IsFalse(eval("abcde", "ee"));
```

Using @Context and Auto-discovery of Fields and Properties

```
var eval7 = m_runtime.Compile<int, bool>("@Context = 1");
Assert.IsTrue(eval7(1));
Assert.IsFalse(eval7(2));

var testData = new TestData {Int64Field1 = 25};
var eval8 = (Func<TestData, int, bool>) m_runtime.Compile(
"Int64Field1 = @arg", typeof(bool),
new Tuple<string, Type>("@Context", typeof(TestData)),
new Tuple<string, Type>("@arg", typeof(Int32)));
Assert.IsTrue(eval8(testData, 25));
Assert.IsFalse(eval8(testData, 26));
```

Handling Infinity and NaN

```
CheckValue(Double.PositiveInfinity, "1.0/0");
 CheckValue(Double.NegativeInfinity, "-1.0/0");
 CheckValue(Double.PositiveInfinity, "1.0/-0");
 CheckValue(Double.NegativeInfinity, "-PositiveInfinity");
 CheckValue(Double.NegativeInfinity, "-1 * PositiveInfinity");
 CheckValue(Double.PositiveInfinity, "-NegativeInfinity");
 CheckValue(Double.PositiveInfinity, "PositiveInfinity * 2");
 CheckValue(Double.PositiveInfinity, "PositiveInfinity + 1");
 CheckValue(Double.NegativeInfinity, "-(PositiveInfinity / 0)");
 CheckValue(Double.PositiveInfinity, "PositiveInfinity / -0");
 CheckValue(Double.NaN, "PositiveInfinity - PositiveInfinity");
 CheckValue(.0, "-0 / -PositiveInfinity");
 CheckValue(true, "1.0/0 = PositiveInfinity");
 CheckValue(true, "-1.0/0 = NegativeInfinity");
 CheckValue(true, "PositiveInfinity > NegativeInfinity");
 CheckValue(Double.NaN, "PositiveInfinity + NegativeInfinity");
 CheckValue(Double.NaN, "PositiveInfinity - PositiveInfinity");
 CheckValue(true, "IsInfinity(PositiveInfinity * 2)");
 CheckValue(true, "IsInfinity(cast(PositiveInfinity, 'single') * 2)");
 CheckValue(true, "IsNaN(PositiveInfinity + NegativeInfinity)");
 CheckValue(true, "IsNaN(PositiveInfinity - PositiveInfinity)");
 CheckValue(false, "NaN = NaN");
 Assert.AreEqual(Single.NaN, Eval<float, float>("@context", Single.NaN));
 Assert.IsTrue(Eval<float, bool>(
        "IsNan(cast(@context, 'double'))", Single.NaN));
 Assert.IsTrue(Eval<Double, bool>(
        "IsNan(cast(@context, 'single'))", Double.NaN));
 Assert.IsTrue(Eval<Double, bool>("IsNan(@context)", Double.NaN));
 Assert.IsTrue(Eval<float, bool>(
        "IsInfinity(@context)", Single.PositiveInfinity));
 Assert.IsTrue(Eval<Double, bool>(
        "IsInfinity(@context)", Double.PositiveInfinity));
 Assert.IsTrue(Eval<Double, bool>(
        "1./0 = @context", Double.PositiveInfinity));
 Assert.IsTrue(Eval<Double, bool>(
        "cast(1./0, 'single') = -@context", Double.NegativeInfinity));
```

Nullability

```
var eval1Result = Eval<int?>("Null");
Assert.IsFalse(eval1Result.HasValue);

Assert.AreEqual(1, Eval<int>("1+cast(Null, 'int32')"));
Assert.AreEqual(1, Eval<int>("1+Convert(Null, 'int32')"));
Assert.AreEqual(1, Eval<int>("1+Null"));
Assert.IsNull(Eval<string>("Null"));
Assert.AreEqual("test", Eval<string>("Null + 'test'"));

var eval4 = (Func<UnboxableNullable<int>, int>)m_runtime.Compile(
    "5 + CASE @arg when 1 then NULL else 25 END",
    typeof(int), new Tuple<string, Type>(
        "@arg", typeof(UnboxableNullable<int>)));

Assert.AreEqual(5, eval4(1));
Assert.AreEqual(30, eval4(-2));
Assert.AreEqual(30, eval4(0.Null()));

eval4 = (Func<UnboxableNullable<int>, int>)m_runtime.Compile(
    "CASE WHEN IsNull(@arg) then 1 else 2 END",
    typeof(int), new Tuple<string, Type>(
        "@arg", typeof(UnboxableNullable<int>)));

Assert.AreEqual(2, eval4(1));
Assert.AreEqual(1, eval4(0.Null()));

eval4 = (Func<UnboxableNullable<int>, int>)m_runtime.Compile(
    "CASE WHEN @arg iS NuLl then 1 else 2 END",
    typeof(int), new Tuple<string, Type>(
        "@arg", typeof(UnboxableNullable<int>)));

Assert.AreEqual(2, eval4(1));
Assert.AreEqual(1, eval4(0.Null()));

eval4 = (Func<UnboxableNullable<int>, int>)m_runtime.Compile(
    "CASE WHEN @arg between 0 and 0 then 1 else 2 END",
    typeof(int), new Tuple<string, Type>(
        "@arg", typeof(UnboxableNullable<int>)));

Assert.AreEqual(2, eval4(1));
Assert.AreEqual(1, eval4(0.Null()));

eval4 = (Func<UnboxableNullable<int>, int>)m_runtime.Compile(
    "CASE @arg WHEN NULL then 1 else 2 END",
    typeof(int), new Tuple<string, Type>(
        "@arg", typeof(UnboxableNullable<int>)));

Assert.AreEqual(2, eval4(1));
Assert.AreEqual(1, eval4(0.Null()));

var eval5 = (Func<string, int>)m_runtime.Compile(
    "CASE @arg WHEN NULL then 1 else 2 END",
    typeof(int), new Tuple<string, Type>("@arg", typeof(string)));

Assert.AreEqual(2, eval5("test"));
Assert.AreEqual(2, eval5(string.Empty));
Assert.AreEqual(1, eval5(null));

Assert.IsTrue(Eval<UnboxableNullable<int>, bool>(
        "isNull( CASE @context WHEN 1 THEN NULL ELSE NULL END)", 0.Null()));
Assert.AreEqual(1, Eval<int, int>(
        "1 + CASE @context WHEN 1 THEN NULL ELSE NULL END", 0));
Assert.AreEqual(1, Eval<int, int>(
        "1 + CASE @context WHEN NULL THEN NULL ELSE NULL END", 0));
```

```
Assert.AreEqual(1, Eval<UnboxableNullable<int>, int>(
        "1 + CASE @context WHEN 1,NULL,2 THEN 1 ELSE NULL END", 0));
Assert.AreEqual(2, Eval<UnboxableNullable<int>, int>(
        "1 + CASE @context WHEN 1,NULL,2 THEN 1 ELSE NULL END", 1));
Assert.AreEqual(2, Eval<UnboxableNullable<int>, int>(
        "1 + CASE @context WHEN 1,NULL,2 THEN 1 ELSE NULL END", 2));
Assert.AreEqual(2, Eval<UnboxableNullable<int>, int>(
        "1 + CASE @context WHEN 1,NULL,2 THEN 1 ELSE NULL END", 0.Null()));


Assert.AreEqual(1, Eval<int, int>(
        "1 + CASE WHEN @context IS NULL THEN NULL ELSE NULL END", 0));
Assert.AreEqual(6, Eval<UnboxableNullable<int>, int>(
        "1 + CASE WHEN @context IS NULL THEN 5 ELSE NULL END", 0.Null()));
Assert.AreEqual(6, Eval<UnboxableNullable<int>, int>(
        "1 + CASE WHEN @context IS NOT NULL THEN NULL ELSE 5 END",
        0.Null()));

Assert.AreEqual("test", Eval<string, string>(
        "IfNull(null, 'test')", null));
Assert.AreEqual(null, Eval<string, string>("IfNull(null, null)", null));
Assert.AreEqual("test", Eval<string, string>(
        "IfNull('test', 't')", null));
Assert.AreEqual("test", Eval<string, string>(
        "IfNull(@context, 'test')", null));
Assert.AreEqual("test", Eval<string, string>(
        "IfNull(@context, 't')", "test"));
Assert.AreEqual(1, Eval<int>("IfNull(1, 2)"));
Assert.AreEqual(1, Eval<UnboxableNullable<int>, int>(
        "IfNull(@context, 2)", 1));
Assert.AreEqual(2, Eval<UnboxableNullable<int>, int>(
        "IfNull(@context, 2)", 0.Null()));

Assert.IsTrue(Eval<bool>("IsNull(cast(Null, 'string'))"));
Assert.IsTrue(Eval<bool>("cast(Null, 'string') is null"));
Assert.IsTrue(Eval<bool>("IsNull(Null)"));
Assert.IsFalse(Eval<bool>("nuLl is noT nuLL"));
Assert.IsTrue(Eval<bool>("Null is null"));
Assert.IsTrue(Eval<UnboxableNullable<int>, bool>(
        "IsNull(@context)", 0.Null()));
Assert.IsTrue(Eval<string, bool>("IsNull(@context)", null));
Assert.IsTrue(Eval<string, bool>("@context is null", null));
Assert.IsFalse(Eval<string, bool>("IsNull(@context)", "test"));
Assert.IsFalse(Eval<string, bool>("@context is null", "test"));
Assert.IsTrue(Eval<string, bool>("@context Is nOt nUll", "test"));

Assert.AreEqual(0.Null(),
        Eval<UnboxableNullable<int>, UnboxableNullable<int>>(
        "@context", 0.Null()));
Assert.AreEqual(1, Eval<UnboxableNullable<int>, UnboxableNullable<int>>(
        "1 + @context", 0.Null()));
Assert.AreEqual(2, Eval<UnboxableNullable<int>, UnboxableNullable<int>>(
        "1 + @context", 1));
Assert.AreEqual(1,
        Eval<UnboxableNullable<int>, UnboxableNullable<decimal>>(
        "null + @context", 1));
```

Implementation Details – CURRENT STATE, MAY CHANGE

E3 implements three operations

- Parsing, produces AST. To parse expression text, E3 uses open-source .NET library called Irony (http://irony.codeplex.com/).
- Analysis, produces Expression. Analyzer takes AST and runtime configuration, and emits Expression tree with explicit data types (e.g. no implicit CLR boxing).
- Compilation, produces an object, which is in fact a constructed Func or Action. Compiler takes an Expression and compiles it into a lambda using argument declarations from CompilerState. The reason to declare return type as an object is to help reduce the runtime overhead from generalization of the runtime interface.

Parser uses an adapted version of SQL-89 grammar (a sample that comes with Irony). The grammar is not exposed via APIs.

Detailed information on each public type is contained within XML comments.

Division by zero, as-implemented now (follows .NET rules at this time)

1) 1.0/0 = PositiveInfinity (a valid Double value, and you can use this literal in E3 expression)
2) -1.0/0 = NegativeInfinity (a valid Double value, and you can use this literal in E3 expression)
3) 1/0 = runtime error, division by zero (because 1 is integer)