ExpressionEvaluator Class for Xojo

Version 1.3.1 beta[1]

Robert S. Weaver
Saskatoon, Canada

1 Fine Print

The **ExpressionEvaluator** class source code is open source. You are free to copy it and use it for any purpose (within **Xojo's** own licensing restrictions, of course). In doing so, you assume all responsibilities for its use, and shall not hold the original author liable for any damages, direct or indirect, caused by its use.

2 Description

ExpressionEvaluator is a **Xojo** class that allows the developer to work with mathematical expressions entered (as text) by the user. The expression syntax is similar to, but not necessarily exactly the same as that of a **Xojo** mathematical assignment statement. The differences are discussed in Section 7.

I have developed its features to suit my own particular projects. At the same time, I've tried to make things as general as possible, since there's no telling what features I or others may need in future projects. Inevitably, the **ExpressionEvaluator** class will not fully address the needs of every developer. But hopefully, it will give a good foundation that can be suitably adapted without too much effort.

The **ExpressionEvaluator** class includes two principal methods: **Parse** and **Eval**.

Parse accepts a mathematical expression as a string argument, performs a syntax check, and if no errors are encountered, parses it into:

- a command list array which is later used for efficient evaluation of the expression by the **Eval** method, using stack operations;
- a symbol table of the variables used in the expression;
- a reformatted version of the input expression with uniform spacing and capitalization;
- a postfix version of the input expression;
- other miscellaneous things.

Eval uses the symbol table information and command list, created by **Parse**, to evaluate the expression. All operations are carried out using double precision floating point numbers, and the result is returned as type **double**.

[1] For revision history, please refer to the last pages of this document.

This two stage process makes the evaluation of the expression much more efficient, because evaluation is far less processing intensive than parsing. This is important for situations where the expression must be evaluated many times for different input values, as would be the case in graphing or optimizer/solver applications. Of course, there is nothing to prevent the developer from running **Parse** and **Eval** together, shielding the end user from the internal goings on.

For further details about these, and other methods and properties, refer to the following sections.

This class was developed using **Xojo 2016r3**, and later tested with **2016r4.1** and **2017r1**. A bug was introduced into release **2016r4.1** (see Feedback Case #46553) where the Auto datatype would incorrectly return boolean and integer values as strings. Two new functions ***AutoToInt()*** and ***AutoToBool()*** were added to the class as a workaround. The bug still exists in the latest release of **Xojo**, but as far as I'm aware, the workaround continues to work, and so this class should work with the latest **Xojo** releases. However, since none of the releases after **2016r3** add new features of any benefit to this class, I will continue to develop it in **2016r3** until further notice, and won't necessarily test it with each new **Xojo** release. If you encounter any incompatibilities with newer **Xojo** releases, please contact me at the email address given in Section 7.5.

3 Speed

First of all, it should be pointed out that everything that is provided in this class can also be accomplished using **Xojoscript**, which will do it faster. However, Xojoscript also has limitations. It is sandboxed with limited access to application resources. The ExpressionEvaluator class, while slower has complete access to all application resources. Admittedly, the choice not to use **Xojoscript** was made because it seemed like a better programming exercise to use traditional expression parsing techniques. Nevertheless, every effort has been made in this class, to evaluate expressions as efficiently as possible, while at the same time making it fairly easy to extend the class with new features as needs arise.

4 Quick Start Tutorial

The **ExpressionEvaluator** class is distributed as part of an example **Xojo** project file, **ExpressionEvaluator.xojo_binary_project**. The project file includes two examples. If you wish to try these before going any further, then skip ahead to Section 5. Otherwise, the tutorial in this section will give you all of the basics for using this class. You can work with the project file, or copy the **ExpressionEvaluator** class from the project file to your own project file.

For each expression to be evaluated, create an instance of the **ExpressionEvaluator** class:

```
Dim myExpEval As new ExpressionEvaluator
```

In this example, let's assume the expression we wish to evaluate is: **`2*x + 7*y`**

Parse the expression:

```
Dim success As boolean = myExpEval.Parse("2*x + 7*y")
```

Parse returns **True** if the expression is successfully parsed, and **False** otherwise. The **Parse** method logs the results of its operations as it parses the expression. If **Parse** returns **False**, you can check the log to help diagnose the error.

```
Dim myErrorLog As string = myExpEval.GetLog
```

If **Parse** is successful, the variables in the expression are added to the symbol table. The symbol table is in the form of two shared property arrays. The first, **VarNames**, is an array of the variable names. The second, **VarValues**, is an array of the values of the variables. When a variable is first created, it is assigned the value **0.0**.

For the above example, the arrays would contain the following elements:

Element No.	varNames	varValues
0	$result	0.0
1	x	0.0
2	y	0.0

The variable **$result** is a default variable automatically created to hold the result of the expression evaluation.

You can also explicitly give a destination variable in typical assignment statement syntax:

```
Dim success As boolean = myExpEval.Parse("z = 2*x + 7*y")
```

This would result in the following variables.

Element No.	varNames	varValues
0	$result	0.0
1	z	0.0
2	x	0.0
3	y	0.0

Note that variable **$result** is always created regardless of whether or not a destination variable has been specified. It will always occupy element **0** of the array. The positions of the other elements are typically the same as the order in which they are created, but this should never be relied upon.

Because the symbol table arrays are shared properties, the variables created by one **ExpressionEvaluator** instance are accessible to all other instances. Hence, the output variable from one expression can be used as an input variable in another expression with no additional programming.

Before evaluating the expression, some values must be assigned to the variables. The variables are assigned using the **SetVariable** method, as follows:

```
myExpEval.SetVariable("x",5.0)
myExpEval.SetVariable("y",9.0)
```

This assigns the value **5.0** to variable **x**, and **9.0** to variable **y**.

Now, to evaluate the expression. simply call the Eval method:

```
myExpEval.Eval
```

This evaluates the expression and stores the result (**73.0**, in this case) in either **$result** or the specified destination variable **z** (as in the second example expression). To get the result of the evaluation, you can use the **Result** method:

```
Dim myResult As double = myExpEval.Result
```

This always returns the result of the last evaluation of the **ExpressionEvaluator** instance, regardless of whether it was assigned to **$result** or to another specified variable.

Combining everything discussed up to this point, our program might look like this:

```
Dim myExpEval As new ExpressionEvaluator
If myExpEval.Parse("z = 2*x + 7*y") then
  myExpEval.SetVariable("x",5.0)
  myExpEval.SetVariable("y",9.0)
  myExpEval.Eval
  myTextField.text = "The result is: "+str(myExpEval.Result)
Else
  myTextField.text = myExpEval.GetLog
End If
```

Suppose we now create two more instances of **ExpressionEvaluator** with the expressions:

```
y = 3*z - 5*w
w = x/y
```

The result of parsing these two expressions yields the following symbol table array contents:

Element No.	varNames	varValues
0	$result	0.0
1	z	73.0
2	x	5.0
3	y	9.0
4	w	0.0

Since **x**, **y** and **z** already exist, only the new variable **w** is added to the arrays. The value of **w** defaults to **0.0**. The values of variables **x**, **y** and **z** are what they would be after using the **SetVariable** method for **x** and **y**, as previously described, and then evaluating the expression: **`z = 2*x + 7*y`**

The value of **$result** hasn't changed, because it has not been assigned as the destination. This is an important distinction between how the method **Result** works and how the variable **$result** is used. Method **Result** will always return the result of the last evaluation of the instance regardless of which, if any, destination variable is assigned. Variable **$result** will only be assigned the result of the evaluation if no other destination variable has been specified.

There is an alternative method for creating new variables. You can use the **SetVariable** method which was previously used to assign values to existing variables. If the variable doesn't already exist, then **SetVariable** creates it and assigns its value.

```
SetVariable(name As String, value as Double)
```

SetVariable can be called before any instances exist, using the syntax:

```
ExpressionEvaluator.SetVariable("myPiVar",3.14159)
```

If a variable with the specified name already exists, it is simply set to the new value.

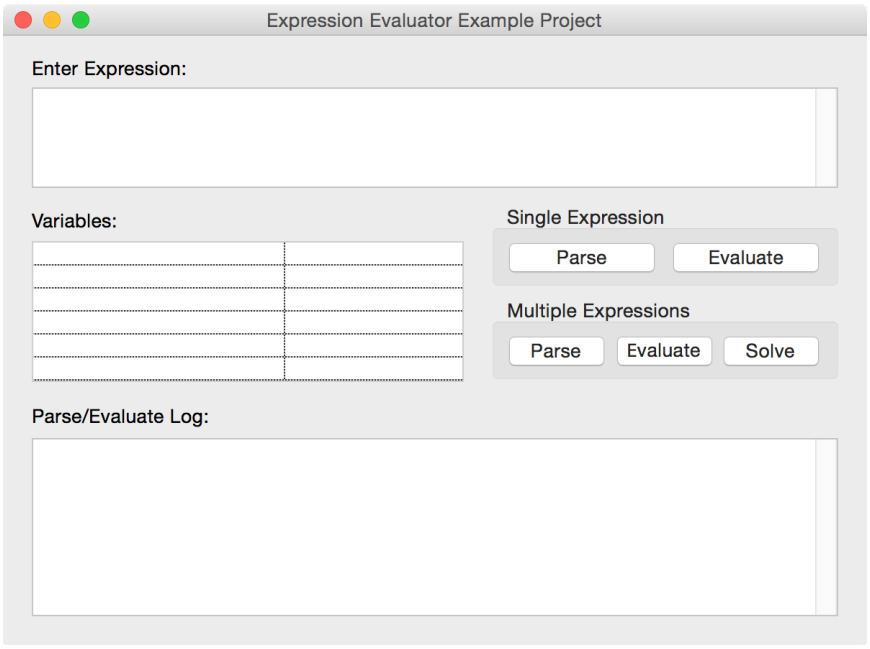
In order to retrieve the value of any variable, use the **GetVariable** method.

```
GetVariable(name As String) As Double
```

This covers the majority of what you need to know in order to use the **ExpressionEvaluator** class. There are a few additional methods and properties which allow for more efficient manipulation of data. They are described in detail in the reference section.

5 Examples

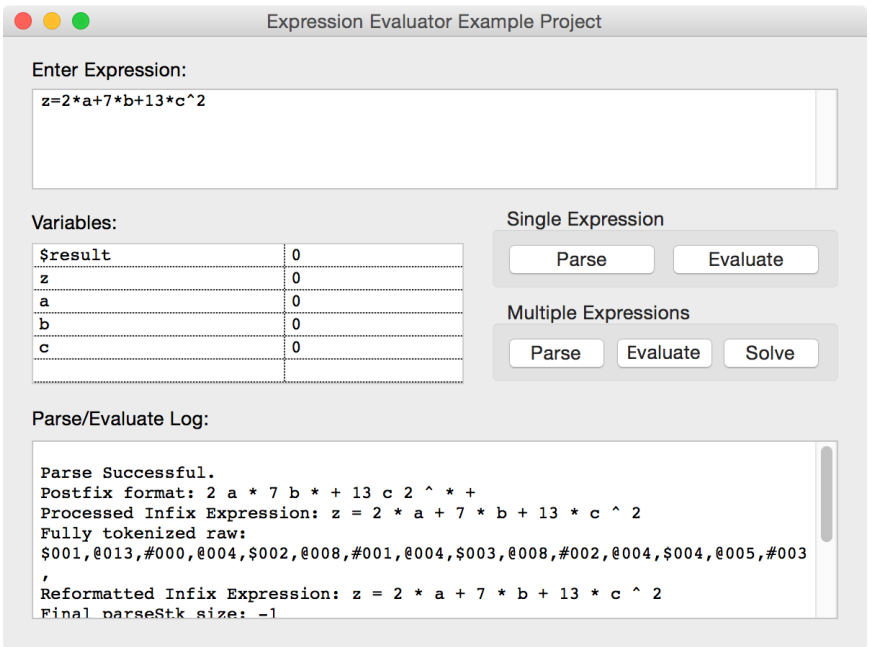
The project file, in which the **ExpressionEvaluator** class is distributed, includes two examples: A single expression evaluation, and a multiple expression evaluation.
When you run the application, you are presented with this window:



We will start with a single expression example. Copy the following expression:

```
z=2*a+7*b+13*c^2
```

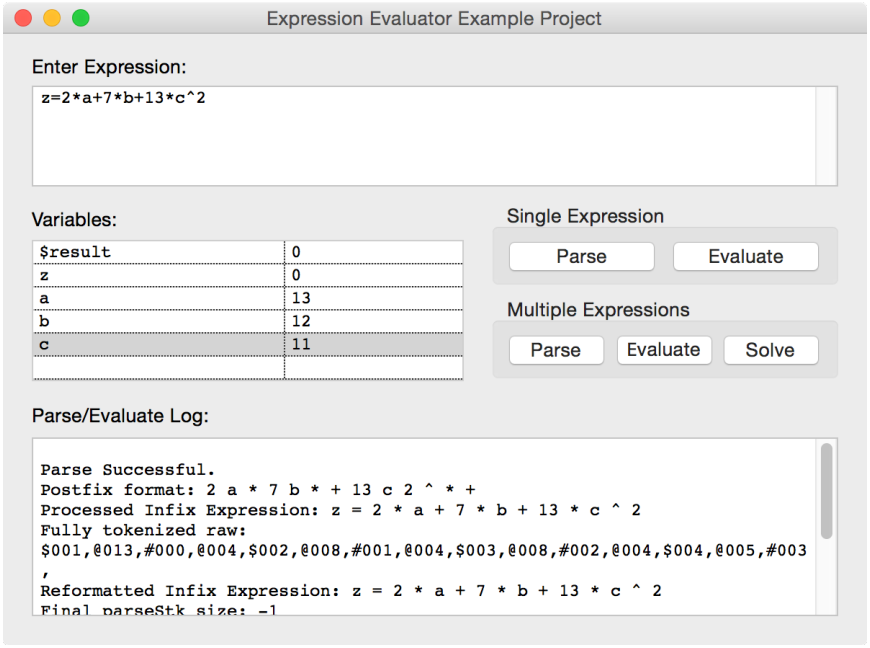
Then, paste it into the **Enter Expression** text area in the application window.
In the **Single Expression** control group there is a pair of buttons. Click the **Parse** button.
The result should look like this:



The **Parse/Evaluate Log**, at the bottom of the window, contains status information that the **Parse** method generated as it parsed the expression. However, all that we care about is the first line that says "Parse Successful."

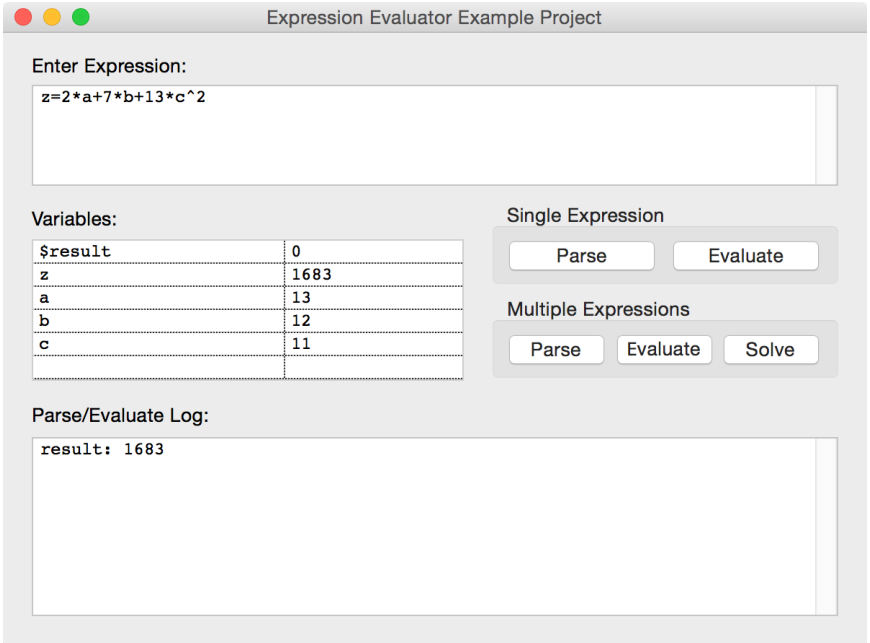
In the **Variables** listbox, all of the variables from the expression are displayed along with their default values of **0**.

Now, enter values in the list box for variables **a**, **b**, and **c**: **13**, **12** and **11** respectively.



Then, click the Evaluate button in the Single Expression control group.

The expression is evaluated, and the result, **1683** is assigned to destination variable **z**.



If you look at the code for the Single Expression Parse and Evaluate buttons, you'll see that it is very straightforward, and uses the methods described in the previous section.

Next, we will look at a multiple expression example.

Let's consider a system of 3 linear equations in 3 unknowns that we wish to solve:

```
2*x + 3*y - z = 14
5*x - 4*y + 2*z = 13
x + y + z = 23
```

A well known method for solving them is the Gauss-Seidel method. It requires that we rearrange the equations so that one of the unknown variables appears by itself on one side of the equals sign. For the first equation, we can move all but the **x** term to the right.

```
2*x = 14 - 3*y +z
```

Now, dividing both sides by **x**'s coefficient **2** we get

```
x = (14 - 3*y + z)/2
```

We do the same for the second equation, except that this time we separate out the variable **y**.

```
y = (13 - 5*x - 2*z)/-4
```

And separating out variable **z** in the third equation.

```
z = 23 - x - y
```

In the Gauss-Seidel method, we simply start with an estimate (or wild guess) for values of the variables. Then, substitute them into the above formula, evaluate each formula, and replace the previous estimate of the variables with the new ones.

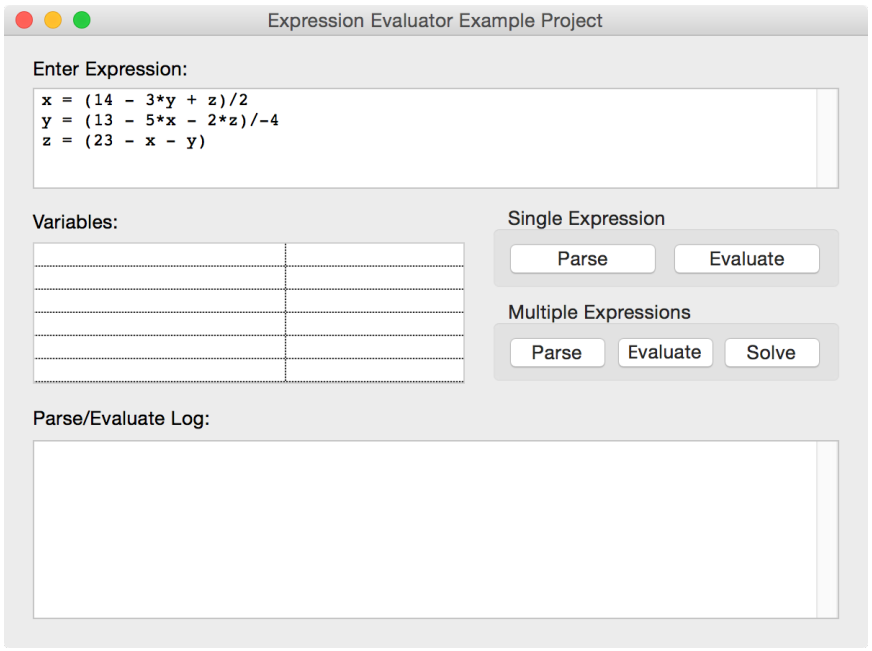
For example, let's say we start with an initial estimate of **0.0**, **0.0** and **0.0** for **x**, **y** and **z**.

Applying these values to the first equation, and evaluating it, we get a new value for **x** of **7.0**. Next we apply the values **7.0**, **0.0** and **0.0** to the second equation, and evaluate it. This gives a new value for **y** of **-11.0**. Applying the values **7.0**, **-11.0** and **0.0** to the third equation, and evaluating it, we get a new value for **z** of **27.0**. If we continue to evaluate each expression in sequence, using the latest calculated values for **x**, **y** and **z**, the numbers should converge to the correct values of the variables. Unfortunately, this doesn't always work. Sometimes the values diverge. To explain why this happens, and how to fix it involves matrix algebra theory, and is beyond the scope of this example. Suffice to say that if we set the new value of the variable to the mean of the old value and the newly calculated value, the values are more likely to converge[2]. So, for the first calculation where we get a new value of **7.0** for **x**, we will take the average of the new value **7.0** and the old value **0.0** to get **3.5**. In the example project, the Gauss Seidel evaluator method handles this automatically. So, the equations may be entered as shown below.

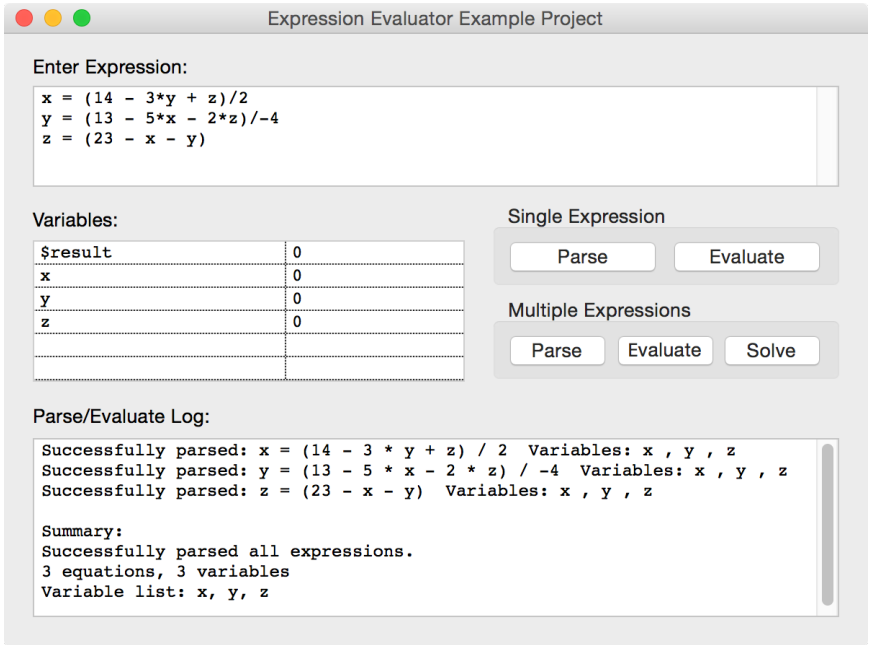
```
x = (14 - 3*y + z)/2
y = (13 - 5*x - 2*z)/-4
z = 23 - x - y
```

[2] The likelihood of convergence improves, but there are still systems of equations which will fail to converge. So, this simple example should not be used as the basis of a rigorous equation solving application.

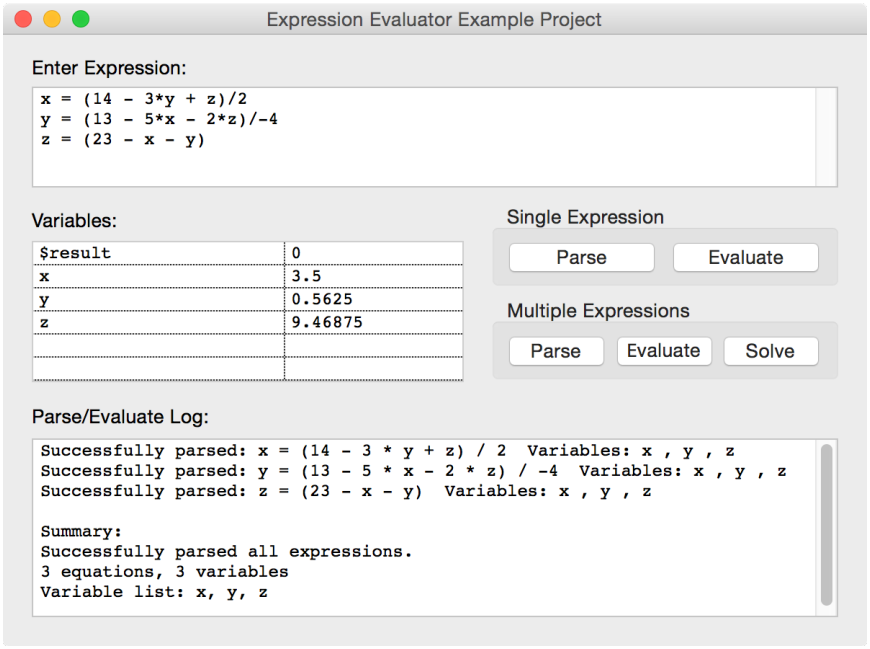
Now, copy the above set of equations, and paste them into the **Enter Expression** text area. It should look like this:



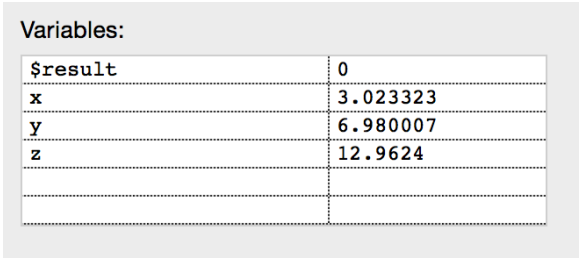
In the **Multiple Expressions** control group, click the **Parse** button.



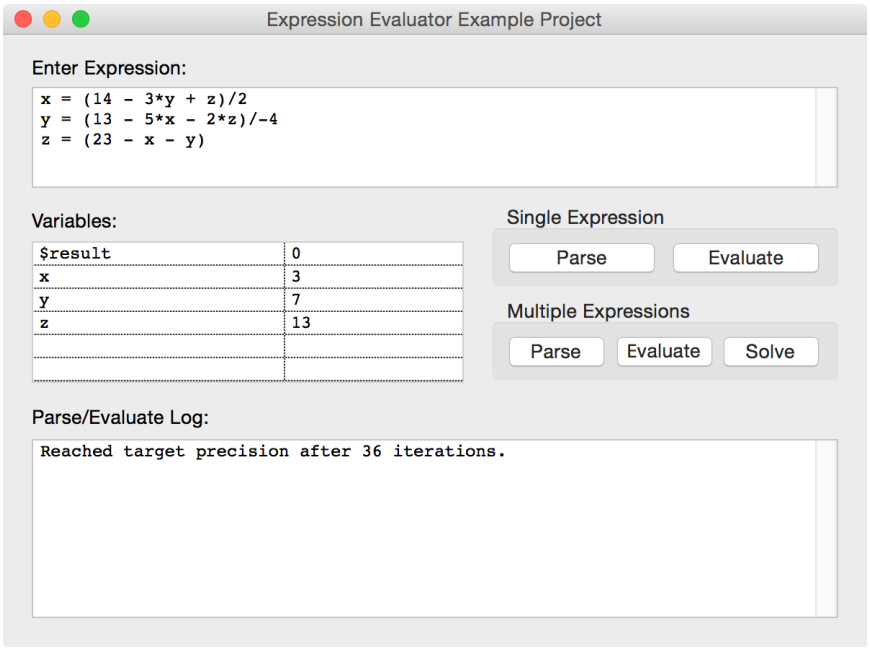
Now, in the **Multiple Expressions** control group, click the **Evaluate** button.



After the first evaluation, the variables **x**, **y** and **z** have the values shown in the listbox in the above screenshot. If you continue to click the Evaluate button, the values start to converge to their exact values. After ten iterations:



Naturally, for a practical application we would use a loop to iterate the evaluations, and use a test to compare previous to current values in order to exit the loop when the change in values drops below a certain threshold. That is what the **Solve** button does. Clicking the **Solve** button gives the following result.



The numbers shown above are the exact solution to the set of equations.

These examples should give a good idea of some of the things that are possible with the **ExpressionEvaluator** class. Have a look at the code in the button action event handlers for implementation details.

6 Reference

The methods and properties of the **ExpressionEvaluator** class are briefly explained below. In the first version of this documentation, I included, for completeness sake, every property and method in the class. I realize that this was a bit of overkill. In recent versions, methods or properties that have been added to the class for internal use only may not be included in the documentation. For information about methods not included here, please refer to the source code. All methods are well commented in the source listing.

6.1 Methods

`Parse(exprString As String) As Boolean`

The **Parse** method is the workhorse function of this class. It syntax-checks, and then parses the **exprString** argument into the command list array **cmdList**. It adds any included variables and constants to the symbol table arrays. As it proceeds with parsing, it adds status and diagnostic information to the **errLog** string, which may be retrieved with the **getLog** method. In addition, **Parse** creates a reformatted version of the input expression and stores it in **infixTxt**, and creates a postfix version of the input expression and stores it in **postfixTxt**. **Parse** also sets the values of several other properties. These are discussed in the applicable property descriptions.

Parse operates as follows:

1. Call the **Init** method which sets up the operator data needed for parsing. If **Init** has previously been called, then the data structures are already in place and it immediately returns.
2. Scan the expression to ensure that it contains no illegal characters, and that any parentheses are properly nested and balanced.
3. Scan the expression for numerical constants formatted in scientific notation, and if found, convert any plus and minus signs in the exponent to escape characters.
4. Scan for operator symbols and replace them with tokens. As a result of this replacement, the expression string is fully delimited with spaces between all tokens and un-tokenized symbols. The expression string is now converted into a string array using the spaces as delimiters for splitting the string.
5. Scan the remaining un-tokenized expression symbols, which should be either variable names or numeric constants. These are replaced with variable and numeric tokens, and these variables and constants are added to the symbol table arrays. At this point, the expression is fully tokenized. The tokens are strings of four characters each. The first character designates the type, and will be **@** for operators, **#** for numeric constants, and **$** for variables. The remaining three characters of the token are a three digit sequence number indicating the specific operator, constant or variable.

6. Scan the tokenized expression to detect situations where the minus symbol "–" should be treated as a unary minus, instead of the subtraction operator. The unary minus operation is distinct from subtraction, because it takes only one operand, while subtraction takes two. Unary minus has a different token assigned. Also, once recognized as a unary minus, the symbol is changed internally from "–" to "~" (tilde).
7. As a result of the previous operations, some of the elements in the expression array, may now be empty. The array is compressed, eliminating the null elements.
8. Check to see if the expression contains the assignment operator = and a destination variable. If so, it strips these out of the expression and sets the appropriate destination variable parameters.
9. Call the Validate method to test whether the expression, as it now stands, is well formed.
10. The expression array is now used to generate and store an equivalent postfix version of the expression array.
11. The postfix expression array is used to generate the operator command list **cmdList**. The evaluation stack **evalStack** is re-dimensioned to the necessary size to handle the forthcoming expression evaluation.
12. If any fatal error has occurred in any of the above steps, the **Parse** method will have exited, returning the value **False**. If the parser has successfully reached this point, it now exits, returning **True**.

`DeTokenize(s() As String) As String`

This is a helper method used by **Parse** to convert an array of tokens into an equivalent string of symbols.

`DetokenSubExpr(s() s() as string,a as integer, b as integer) As String`

This is a helper method used by **Validate** to convert a subset of the array of tokens **s**, ranging from index **a** to index **b**, into an equivalent string of symbols.

`Eval`

Evaluates the expression which has previously been parsed by the **Parse** method. It does this by calling, in sequence, all of the operator methods (the **X_methods**) contained in the **cmdList** array. The result of the evaluation is stored in the expression's destination variable if one has been assigned. Otherwise the result is stored in the default **$result** variable.

`FreeMem`

Frees up memory by clearing all of the parsing data arrays used in the parsing process. This may be called when no further parsing of expressions is required. **Eval** does not use any of these data arrays. And so, previously parsed expressions can still be evaluated. If subsequent expressions need to be parsed, the data arrays will automatically be rebuilt. Hence, there is no harm in calling **FreeMem**. However, the

rebuilding of the parsing data arrays does take time. So, it's not advisable to call **FreeMem** between each expression parsing.

`GetLog As String`

Returns the current diagnostic/status log data string.

`Init As boolean`

Compiles a dataset of the X_methods, their names and attributes. These data are used by the parser to determine operator names, operator precedence, whether the operator is unary or binary, etc. **Init** is called automatically by the **Parse** method as necessary. **Init** should always return **True**. If it returns **False**, it indicates that one or more **X_Methods** have bad or missing attributes.

`OutputValue As Double`

Returns the value of the expression's destination variable. This may not be the same as the result of the last evaluation of the expression, because some other code may have altered the variable's value after the expression's evaluation. See also the **Result** method.

`PrecedenceInp(token As String) As Integer`

Returns the Input Precedence of the operator specified by **token**.

As the expression is converted from infix to postfix, the operators are pushed onto and popped off of the parse stack, according to the operator's precedence relative to the precedence of the other operators in the expression. **PrecedenceInp** is used for the operator prior to being pushed onto the stack. After the operator is on the stack, **PrecedenceStk** is used. These two different precedence values account for correct handling of parentheses and left and right associative operators.

`PrecedenceStk(token As String) As Integer`

Returns the Stack Precedence of operator specified by **token**. See the description above for the **PrecedenceInp** property, explaining the purpose of the two different precedence values.

`Rank(token As String) As Integer`

Returns the rank of an operator. This value indicates how the associated operator affects the number of items on the evaluation stack. Binary operators decrease the number of stack items by one. Unary operators leave the number of items unchanged. Variable push and Constant push operations increase the number of items by one.

`Result As Double`

Returns the result of the last evaluation of the expression. This always returns the result of the last evaluation of the expression regardless of whether some other code has changed the value of the destination variable, because this method retrieves the value from the expression's evaluation stack which is not accessible to other code. See also the **OutputValue** method.

`TokenToSymbol(token As String) As String`

Returns the symbol (operator, variable or constant) that corresponds to the token string.

`Validate(tkExpr() As String) As Boolean` (New, as of v1.1.0b)

Checks the tokenized infix expression **tkExpr** to determine if it is well formed. Returns **True** if valid and **False** otherwise. This is called by **Parse**.

`X_Methods`

These are the methods whose names begin with **X_** and which are invoked to perform the individual arithmetic operations on the evaluation stack when the expression is being evaluated. They have no call and no return parameters. They directly access and modify **evalStack** , **stkPtr**, and the **varValues** array. They also read **cmdPtr**, the **param** array and the **constants** array but do not change them.

The following chart shows a complete list of the current **X_methods**:

Operation	Symbol	Method	Prec.	Assoc.	Rank
Add	+	X_Add	4	0	-1
Subtract/Unary Minus	-	X_Sub	4	0	-1
Multiply	*	X_Mul	5	0	-1
Divide	/	X_Div	5	0	-1
Raise to Power	^	X_Pwr	6	1	-1
Unary Minus	~	X_Neg	8	1	0
Integer Divide	\	X_Idv	5	0	-1
mod (remainder)	mod	X_Mod	5	0	-1
Abs	abs(	X_Abs	7	1	0
Ceiling	ceil(	X_Ceil	7	1	0
Floor	floor(	X_Floor	7	1	0
If	if(	X_If	7	1	-2
Integer	int(	X_Int	7	1	0
Max	max(	X_Max	7	1	-1
Min	min(	X_Min	7	1	-1
Square Root	sqrt(	X_Sqt	7	1	0
Exponential	exp(	X_Exp	7	1	0
Logarithm (Natural)	log(	X_Log	7	1	0
Sine	sin(	X_Sin	7	1	0
Cosine	cos(	X_Cos	7	1	0
Tangent	tan(	X_Tan	7	1	0
Arcsine	asin(	X_Asin	7	1	0
Arccosine	acos(	X_Acos	7	1	0
Arctangent	atan(	X_Atn	7	1	0
Arctangent-2 (2 argument version)	atan2(	X_Atn2	7	1	-1
Complete Elliptic Integral 1st kind K	ElipK(	X_ElipK	7	1	0
Complete Elliptic Integral 2nd kind E	ElipE(	X_ElipE	7	1	0
Complete Elliptic Integral K—E	ElipKE(	X_ElipKE	7	1	0
Equals Comparison	==	X_EQU	3	0	-1
Not Equals Comparison	<>	X_NEQ	3	0	-1
Less Than or Equals Comparison	<=	X_LEQ	3	0	-1
Greater Than or Equals Comparison	>=	X_GEQ	3	0	-1
Greater Than Comparison	>	X_GRT	3	0	-1
Less Than Comparison	<	X_LES	3	0	-1
Boolean And	AND	X_AND	2	0	-1
Boolean Or	OR	X_IOR	1	0	-1
Boolean Not	NOT	X_NOT	8	1	0
Boolean Exclusive Or	XOR	X_XOR	2	0	-1
Push Constant Pi	pi	X_Pi	9	0	1
Push Variable	$	X_Var	9	0	1
Push Constant	#	X_Con	9	0	1
Assignment*	=	X_STO	0	0	0
Comma Delimiter*	,	X_DLM	1	0	0
Left Parenthesis*	(		10	20	0
Right Parenthesis*	)		0	0	0

* Note that while the left and right parentheses are included in the table, they have no associated X-methods, because these symbols disappear during the parsing process. The left parenthesis precedence value is automatically determined according to the other operator precedence values, to ensure that it always has the highest precedence. And so, its value may change from what is shown here if higher precedence operators are added to the list.

In addition, while the assignment operator and comma delimiter are associated with the **X_STO** and **X_DLM** methods, these methods are never invoked. Storing the evaluation result is handled directly by the **Eval** method, and commas are removed during the parsing process.

There is an explicit unary minus operator shown in the table. The user isn't required to use this, and it may or may not be included in the list of valid characters kValidChars, because the regular minus sign is interpreted according to the expression context to determine whether it's being used as a unary or binary operator. The Parse function internally converts the minus sign to the the unary minus symbol ~ (tilde) when it determines that it is being used as unary operator. This simplifies later processing. However, if it is intended to display the infixTxt string to the user, then it may be wise to first replace all occurrences of ~ with –.

These methods all have attributes assigned as shown in the table: **Symbol** As String, **Precedence** As Integer, **Associativity** As Integer, **Rank** As Integer.

The **ExpressionEvaluator** class has been set up in such a way that it may be easily extended to include additional mathematical operations merely by adding new **X_Methods**, with their included attributes. No other changes are required in the class. The **Init** method locates the **X_methods** and extracts all necessary information to implement them.

6.2 Shared Methods

```
ClearVariables
```

Deletes all variables from the symbol table, except for **$result** (and **$result** is set to 0.0). If existing **ExpressionEvaluator** instances are to be evaluated again, then they must be re-parsed, to regenerate their variables.

```
ExistsVariable(s As String) As Boolean
```

Returns **True** if variable **s** exists in the parsed expression, **False** otherwise.

```
Filter(s As String, filterChars As String, mode As Boolean) As String
```

If **mode** is **True**, the returned string will be equal to input string **s** excluding the characters that are **not** in **filterChars**. If **mode** is **False**, the returned string will be equal to input string **s** excluding the characters that **are** in **filterChars**.

This method uses **Regex**, and therefore **filterChars** should not include any **Regex** meta-characters unless they are escaped with a backslash. Otherwise, results will be unpredictable.

```
GetVariable(varName As String) As double
```

Returns the value of the variable which has the name **varName**.

`GroupVariables(expArray() As ExpressionEvaluator, gvList() As Integer)`

Returns in **gvList** a set of variable indices currently in use by the group of **ExpressionEvaluators** in **expArray**.

`SetVariable(varName As String, varValue As double)`

The value of the variable having the name **varName** is set to **varValue**. If the variable doesn't exist, it is created, and the value is then set to **varValue**.

`ValidVarName(varName As String) As Boolean`

Checks the syntax of the **varName** string argument to determine if it is a properly formed variable name. If it is valid the method returns **True**. Otherwise it returns **False**.

6.3 Properties

`cmdList() Array of Introspection.MethodInfo`

This is the list of pointers to the operator methods, arranged in proper execution order, which is used by the **Eval** method to evaluate the expression. The **cmdList** data is created by the **Parse** method.

`cmdListSize Integer`

This is the **Ubound** of the **cmdList** array. Its value is set by the Parse method.

`cmdPtr As Integer`

Used as an index into the **cmdList** array when the expression is being evaluated. It indicates the current command being executed in the command sequence.

`constants() Array of Double`

The values of the constants that originally appear as text in the expression parsed by the **Parse** method.

`destVarIndex As Integer`

Index of the expression's destination variable in the varNames and varValues arrays. The destination variable is the variable that appears to the left of the assignment operator in the expression.

`errLog As String`

This holds all of the diagnostic and status messages generated by the Parse, Init and Eval methods.

`evalStack() Array of Double`

This is the operations stack used for the evaluation of the expression.

`infixTxt As String`

This is a copy of the string expression argument passed to the Parse method for parsing. After Parse fully tokenizes the expression. infixTxt is updated with a reformatted version of the expression which has uniform capitalization and spacing. infixTxt is never used by any of the class methods. It is for use by the developer for display purposes.

`isParsed As Boolean`

If the expression has been successfully parsed, this it set to **True**. Otherwise, it is **False**. This is checked by **Eval** before it attempts to evaluate the expression.

`param() Array of Integer`

This is a companion to the **cmdList** array. It contains the index of the parameter to be used by the operator that is referenced in the **cmdList** array. Most operators require no parameters. For them the value in the **Param** array is zero. For the **X_Con**, and **X_Var** operators, the **Param** array holds the index to the corresponding constant or variable in the respective **constants** and **varValues** arrays.

`postfixTxt As String`

This is a postfix version of the infix expression parsed by the Parse method. Each symbol in the **postfixTxt** string corresponds to an operation in the **cmdList** array. **postfixTxt** is never used by any of the class methods. It is for use by the developer, if there is a need to convert infix to postfix, or for display purposes.

`stkPtr As Integer`

Index to the current top of the **evalStack**. Used by the **Eval** method and all of the **X_methods** for writing to and reading from the evaluation stack when the expression is being evaluated.

`varList() Array of Integer`

Array of indices to the **varNames** and **varValues** arrays for all of the input variables used by the expression. The destination (output) variable is not included in the array unless it is also used as an input. The destination index is in **destVarIndex**.

6.4 Shared Properties

`constantIndex As Integer`

The index into the **prsXXX** arrays for the Constant Push **X_Con** method data.

`prsAssociativity() Array of Integer`

Left/Right associativity data for the corresponding operators. A value of **0** means the operator is left associative. A value of **1** means the operator is right associative.

`prsOpMethod() Array of Introspection.MethodInfo`

Array of MethodInfo for the **X_method** operator methods.

`prsPrecedence() Array of Integer`

Array of precedence values for the operators.

`prsPrefixOp() Array of Boolean`

Array of values identifying which operators are prefix type functions, such as **sin**, **cos**, **exp**, **log**, etc.

`prsRank() Array of Integer`

Array of rank values for the operators. The rank indicates how operator affects the number of items on the stack. A value of **-1** indicates that the operator reduces the number of items by one (typical of binary operators). A value of **0** indicates that the operator leaves the number of items unchanged (typical of unary operators). A value of **+1** indicates that the operator increases the number of items by one (typical of Constant and Variable Push operations).

`prsSymbol() Array of String`

Array of symbol strings for the corresponding operators.

`tknLeftPar As String`

The token used for the Left Parenthesis

`tknMinus As String`

The token used for the Subtraction operator

`tknPlus As String`

The token used for the Addition operator

`tknPOPmInfo As Introspection.MethodInfo`

The token used for the Stack Pop operator

`tknRightPar As String`

The token used for the Right Parenthesis

```
tknSTOmInfo As Introspection.MethodInfo
```

The token used for the Stack Store operator

```
tknUnaryMinus As String
```

The token used for the Unary Minus operator

```
varIndex As Integer
```

The index into the **prsXXX** arrays for the Variable Push **X_Var** method data.

```
varNames() Array of String
```

Array of all of the variable names that have been created by the **Parse** and **SetVariable** methods.

```
varValues() Array of Double
```

Array of the variable values corresponding with the variable names in **varNames**.

7 Miscellaneous Notes

Since this is open source, and may be freely modified, I've only included what I consider the most basic features, at least for the time being.

For the current release, there is direct access to several properties, rather than protecting them and using **get** and **set** methods. I may change this in the future. In the meantime, use with caution.

7.1 Syntax Peculiarities

The syntax rules for the **ExpressionEvaluator** class are slightly different from **Xojo's** math expression syntax. One reason for this is that I am obviously not privy to how **Xojo** implemented their own expression evaluator, and so I cannot guarantee this one to be the same. Another reason for this is to avoid the ordeal of disambiguation of certain operators. For that reason I chose to use == as the equals comparison operator (à la Java) to distinguish it from the = assignment operator. (In an earlier version of this class, I used := for assignment and = for comparison. To understand the difficulties that ambiguous operators create, have a look at all of the code in the **Parse** method that's used just to distinguish whether the minus sign is intended to be a subtraction operator, a unary minus operator, a leading character of a numerical constant, or an embedded exponent sign in a numerical constant formatted as scientific notation.

Unlike **Xojo** expressions, **ExpressionEvaluator** variable names are allowed to begin with the underscore character.

The fact that there are slight differences between **ExpressionEvaluator** syntax and **Xojo** syntax, should not matter as far as the end user is concerned, since the end user shouldn't be expected to know the internal details of how **Xojo** works.

7.2 Variable Handling

One may ask why variables were implemented with arrays rather than as a dictionary using the variable name as the key. The reason is that the use of arrays is faster and simpler when the expression is evaluated (especially when evaluated many thousands of times), and execution speed was considered most important.

7.3 The Double Data Type

The only data type used by the **ExpressionEvaluator** class when evaluating expressions, is type **Double**. Since there are boolean comparison operators included in the class, some explanation of their operation is in order. Where a **Boolean** value is expected as an

argument to a **Boolean** operator, any non-zero **Double** value is interpreted as **True**, and zero values are interpreted as **False**. The result of any operation (**Boolean** or **Comparison**) that returns a **Boolean** value, will return **1.0** for **True**, and **0.0** for **False**. This is similar to the operation of older versions of the BASIC language which work only in floating point.

Currently in development are versions of the **ExpressionEvaluator** class which use Robert Delaney's BigFloat and BigComplex data types.

http://delaneyrm.com/fpPlugin.html

These are working but have not been tested exhaustively. If you are interested in using them, please contact me at the email address below.

7.4 The LogEntry Diagnostics

The **LogEntry** method serves two purposes.

Firstly, it assembles diagnostic information for the end user to help diagnose syntax errors in the input expression, and other related user errors.

Secondly, it provides debugging information to the developer as the class is further developed, or obscure bugs are discovered. For that reason, if you look through the source code you will see quite a few **LogEntry** function calls that have been commented out. I chose not to delete these from the code, because they become useful again when things go awry. Of course, you can add, modify or delete **LogEntry** calls to suit your own particular application. In some future release, I intend to add the ability to programatically set how verbose the LogEntry is.

7.5 User Feedback

If you have any comments, questions or bug reports regarding either the software or the documentation, I would appreciate receiving it. You can contact me directly by email:
`feedback@electronbunker.ca`

Revision History

2017-01-04 - Version 1.0.0 Beta

First release.

2017-01-07 - Version 1.0.1 Beta

Bug fix	Resolved difference between 2016r3 and 2016r4.1 in how they handle Auto data type. This caused application to crash when compiled in 2016r4.1. This is a verified bug in 2014v4.1 (Feedback Case #46553)

2017-01-26 - Version 1.1.0 Beta

Change	Parentheses are now required around the arguments of functions. This change was instigated, partly by the bugs mentioned below, and partly to make it easier to deal with functions having more than one argument (and of course for better consistency with other software). In handling functions, a new boolean attribute, **Prefix**, has been added to the X_methods. It is set to **True** for function type operators, and **False** for all others. This attribute is loaded into a new boolean array shared property **prsPrefixOp** by the **Init** method.
Change	Functions having more than one argument are now supported. As a result, new functions **Max**, **Min** and **Atan2** have been added. Currently, the number of arguments is fixed by the **Rank** attribute. Consequently, a variable number of arguments is not yet supported, so **Max** and **Min** must take exactly two arguments.
Change	Some minor changes have been made to the user interface in the example project. The **Evaluate** and **Solve** buttons are now disabled until an expression is successfully parsed. This prevents a user from inadvertently trying to evaluate an expression that hasn't yet been parsed. Also, there are some changes in the Variables Listbox behaviour when tabbing from cell to cell, or clicking on cells.
Bug fix	If an expression had no explicit destination variable assigned, and the first symbol was a function such as sin, cos, etc., then parse would give an error even though the expression was valid.
Bug fix	Incorrect precedence values caused unary minus to be parsed out of order when immediately followed by a prefix function (e.g., sin, abs, etc.).
Bug fix	In certain cases, invalid infix expressions could go undetected and would be parsed into an executable command list. It had previously been assumed that any syntax errors that weren't caught by the scanner and tokenization process would be caught in the postfix conversion. Apparently, this was a naïve assumption. So, a new **Validate** method has been added which checks that the final tokenized infix expression is well formed prior to the postfix conversion.

2017-02-07 - Version 1.2.0 Beta

Change	Added three elliptic integral functions ***K***, ***E*** and ***K–E***.
Bug fix	Functions that push a constant on the stack (e.g., pi) had caused the Validate function to return False.

2017-02-14 - Version 1.2.1 Beta

Change	Temporary operands within the X_Methods have been removed and replaced by a shared property. This eliminates the allocation and deallocation of the variables whenever the X_Method is executed, which should make it slightly faster. This also makes it simpler for future upgrades to larger floating point numbers (such as 128 bit floating point) once Xojo is able to handle them.

2018-09-12 - Version 1.3.0 Beta

New Feature	Added functions ***Ceil()***, ***Floor()*** ***Int()*** and ***If()***.
New Feature	Added new shared method ***ExistsVariable()***.
Bug Fix	Resolved issue where a unary minus would be interpreted as a subtract operator, and flagged as an error, if it occurred at the beginning of an expression which didn't have an assignment variable.

2018-09-15 - Version 1.3.1 Beta

Compatibility Issue	Added a workaround to handle a XOJO 2018r2 bug which corrupts the attribute values of the X_methods. **ExpressionEvaluator** class appears now to work with XOJO 2018r2.