



Articles » General Programming » Algorithms & Recipes » Math

An extensible math expression parser with plug-ins

Mathieu Jacques, 13 Mar 2008

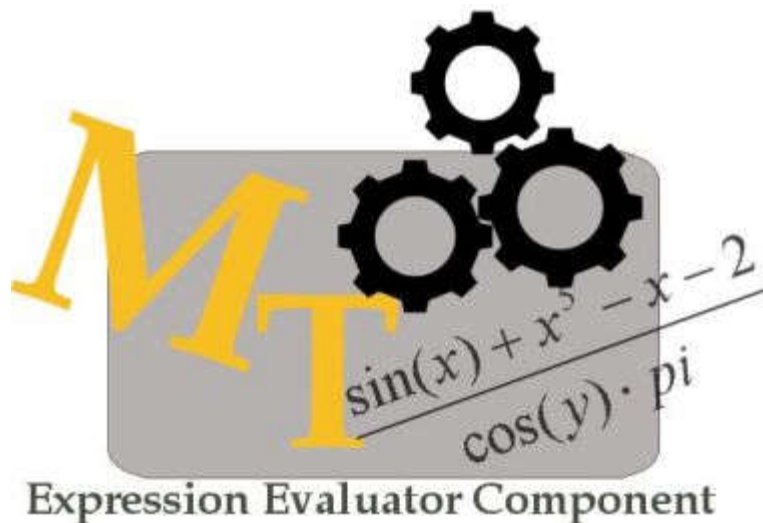
Design and code for an extensible, maintainable, robust, and easy to use math parser.

[Download demo application - 1.05 MB](#)

[Download source - 413.2 KB](#)

[Download MTParserLib's Doxygen documentation- 652.89 KB](#)

[Download MTParserCOM DLL file - 908.04 KB](#)



Introduction

Mathematical expressions are used by almost all applications. In fact, they are the foundation of programming languages. When you type a program, you are actually writing tons of math formulas. A program is static by nature, that is, it cannot be changed while executing. For example, a program written in C++ must be recompiled each time you change a single line of code. Hence, imagine that one math formula has to be modified in order to accommodate a change in the user environment. Do you like the idea of recompiling all your code just for one little formula change? There lies the usefulness of a math expression evaluator since it allows runtime formula evaluations and modifications without the need to change even one line of code. The trick is to handle math formulas as data

instead of program instructions. The evaluator takes formulas as strings, compiles them on the fly, and computes the results that can be used transparently by your application.

Mathematic software is a mature field, and there are many excellent libraries available at minimal fees. Providing enhanced math functions and algorithms is thus not the primary mission of this parser. Instead, the focus is on allowing existing code to be reused in the context of run-time formula evaluation. For this purpose, the parser design defines a framework where existing functions can elegantly be integrated. Plug-in is the privileged way of extending the parser and, if you've developed one, I invite you to send me your component.

There are two parser versions: a native C++ library, and a COM component. The C++ library is highly extensible and fast, but its use is limited to the C++ language only. In counterpart, the COM component is easier to integrate in a project, and many programming languages can use it. COM plug-ins can be used to extend both the C++ library and the COM component.

The article is structured as follows:

1. [Features](#)
2. [License](#)
3. [Design Discussion](#)
 - [Design Patterns](#)
 - [The Parser Design](#)
4. [Algorithms](#)
 - [The Parsing Algorithm](#)
 - [The Evaluation Algorithm](#)
5. [Using the C++ Library and the COM Component](#)
 - [Include the Parser in your Project](#)
 - [Create a Parser Object](#)
 - [Initialize the Object using the Default Configuration](#)
 - [Initialize the Object using Another Object Configuration](#)
 - [Evaluate Mathematical Expressions](#)
 - [Evaluate an Expression and Get the Result Immediately](#)
 - [Evaluate an Expression Multiple Times](#)
 - [Manage Variables](#)
 - [Define a Variable](#)
 - [Define Only the Used Variables](#)
 - [Extend the Parser](#)
 - [Define a Constant](#)
 - [Define a Custom Function or Operator](#)
 - [Define a Macro Function](#)
 - [Load a Plug-in](#)
 - [Load all Available Plug-ins](#)
 - [Localize the Parser](#)
 - [Initialize the Localizer](#)
 - [Localize Error Messages](#)
 - [Localize Item Documentation](#)
 - [Localize the Parser Syntax](#)
 - [A Little Localization Note](#)
6. [Handle Errors](#)
7. [Using COM Plug-ins](#)

- [Create a Plug-in](#)
- [Implement a Plug-in](#)
- [Register a Plug-in](#)
- 8. [Files to Distribute](#)
- 9. [Demo Applications](#)
- 10. [Other Math Parsers](#)
- 11. [Conclusion](#)
- 12. [References](#)
- 13. [History](#)

Features

Here are the main features:

- **User-defined Functions and Operators**

C++ functions and operator classes can be defined to extend the parser language. Usually, these classes will be packaged as a plug-in to allow run-time extension.

- **User-defined Variables and Constants**

A variable is a special symbol that is replaced by a value when evaluated, but unlike a constant, its value can be changed. It can be used to make parameterized formulas like " $3t+10$ ", where t is the time variable. An unlimited number of variables can be defined at run-time.

A constant is a special symbol that is replaced by a value when evaluated. Using constants instead of values helps to make math formulas clearer, especially for well known values. The parser can also do optimizations since it knows that these values are constants. Examples of constants are pi (3.14159...) and e (2.71...). An unlimited number of constants can be defined at run-time.

- **User-defined Macros**

A macro function is helpful to simplify the writing of frequently used expressions. For example, if you often use the Euclidean distance defined as $\sqrt{x^2+y^2}$, then it may be a good idea to define a macro named "euc(x,y)". A macro can use any defined operators, functions, constants, and macros. An unlimited number of macros can be defined at run-time.

- **Batch Evaluate**

To achieve very high performance, you can evaluate an expression multiple times, using a different set of values each time, in one function call. This way, you save expensive function call time, especially when using the COM library version. The result is that a client developed in C# using the COM library version performs at almost 70% of an unmanaged C++ application!

- **Plug-in Extension**

A plug-in allows run-time extension of the parser language by adding functions, operators, and constants. Multiple plug-ins can be loaded based on your user needs, thus allowing run-time customization of your application and enabling component based release management policies (users only pay for the features they use). Another benefit of a plug-in architecture is that third-party components can easily be integrated with your application. It is interesting to know that plug-ins are as fast as native C++ functions and operators.

- **International Considerations**

Today's international software market demands localized applications. This component has been thought for it since the beginning, and thus the localization task is simplified.

- **Localized error messages**

Parameterized error messages are stored in an XML file for each language. All errors come with supplementary fields, allowing very informative messages to be produced. There is no internal message: all messages can be localized.

○ Localized documentation

Documentation can be provided with functions, operators, constants, and variables to help users figure out how to use them. The documentation texts are stored in an XML file.

○ Configurable syntax

The decimal and the function argument separator characters can be modified at run-time to customize the syntax according to your user preferences.

○ Unicode-ready

All strings are handled as Unicode. This includes variables, constants, operators, and function names. However, using a simple compilation option, you can still enable ANSI strings.

• Many Predefined Operators and Functions

More than 16 operators and 30 functions. Boolean, trigonometric, and numerical approximation functions are included. This is only the beginning since plug-ins can be loaded to extend the parser language.

Operators:

+	Addition
-	Subtraction and unary minus
*	Multiplication
/	Division
^	Power
%	Modulo
&	Logical AND
	Logical OR
!	Logical NOT
>, >=	Greater or equal
<, <=	Smaller or equal
!=	Not equal
==	Equal

General functions:

abs	fact	rand()
acos	floor	rand(min, max)
asin	hex	round
atan	if	sin
avg(x,y,z,...)	isNaN	sinh
bin	log	sqrt
ceil	log10	sum(x,y,z,...)
cos	max(x,y,z,...)	tan
cosh	min(x,y,z,...)	tanh

Numerical Approximation functions (with the Numerical Approximation plug-in):

derivate(expr, var, point)	Compute a numerical approximation of the derivative of the expression with respect to one variable at the specified point
trapezoid(expr, var, a, b, [step=0.1])	Compute a numerical approximation to the integral between a and b using the trapezoid rule

Find the variable value that yields the desired result

`solve(expr, var, result, [vo=0], [tol=0.01], [maxIter=100])` using Newton's numerical approximation method

Date/Time functions (with the Date plug-in):

<code>date</code>	<code>nowdate</code>
<code>datevalue</code>	<code>nowtime</code>
<code>day</code>	<code>second</code>
<code>hour</code>	<code>time</code>
<code>minute</code>	<code>weekday</code>
<code>month</code>	<code>year</code>

• User-defined Function Compilers

In order to allow special syntax to be used, functions can compile their arguments the way they want. Most functions are okay with numbers only, but some complex functions need more flexibility. For example, the **solve** function takes a math expression as its first argument, and optional arguments can even be provided.

• Informative Error Messages

Errors are reported via an exception structure, with detailed fields allowing very informative error messages to be produced. The exception structure fields contain information about the “what” and the “where” of each error.

• Very Fast

Fast algorithms are used to assure excellent performance. Some techniques used are expression compilation, constant expression folding, and binary search.

A large feature list doesn't necessary make a good parser. Quality is also very important. So, which qualities should a first-class math parser have? If you want to use it in a real-time application, then performance is a must. And since real-time applications often don't use Windows, the parser must be portable to other platforms. If this is your case, then don't lose your time with this parser, since it is not the “fastest parser in west” and nor is it portable.

However, this parser has some very interesting qualities! A little performance has been traded for extensibility, code maintainability, ease of use, and robustness. Each of these qualities has a negative impact on performance, but it pays off other ways:

- **Extensibility** allows you to easily add operators and functions using clean, object oriented code, and without having to modify existing code. This will avoid introducing subtle bugs and wasting your time debugging the broken parser. Another good thing is that you don't need to understand the inner parser logic to add operators and functions.
- **Maintainability** in the case of an open source library is certainly desirable! There is no obscure code for the performance sake. Dark code has the characteristic that nobody wants to modify it but for the simplest change. Maintainability is defined by the ease of understanding, ease of modifying, ease of testing, and ease of forecasting modification side effects.
- **Ease of use** assures that it will not be a pain in the neck to integrate the parser with your project. This will allow you to concentrate on your application and not waste time understanding a complicated parser interface. Another aspect is internationalization: it can be very time consuming to internationalize a library that has not been thought for this since the very beginning.
- **Robustness** means that your application will not crash because of some missing validation code. And as you know, users are good at finding these little holes...! This also means that the parser will give the correct results and do the necessary validations to ensure result validity. For example, the parser shall not accept syntax that could have side effects, like: “5.2.3 + 3.2.3”. These are two very different things: to work, and to work as expected.
- And of course, the parser is **pretty fast**! But as I said, this is not the top priority, and thus, sometimes choices are made that hinder performance (not without any remorse...!).

Don't get me wrong: this parser is not perfect in regard to all the above qualities. But these qualities are what I care

for and will continue to steer the parser toward.

License

Free to use for non-commercial purposes. If you want to use this parser in a commercial product, please contact me with the following information:

- your programming language
- features useful to you (see the [feature list](#))
- short description of how the parser helps you

In fact, even if you use this library in a personal project, I would really appreciate hearing some words from you! So feel free to share your experience.

Design Discussion

This section presents the overall design and the tradeoffs.

Design Patterns

The following design patterns have been used: Façade, Prototype, Abstract Factory, and State.

Façade

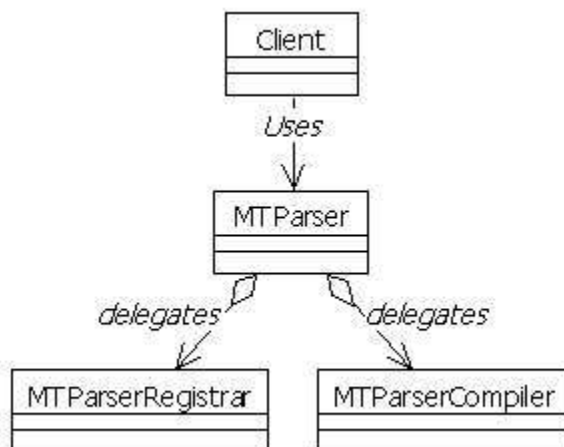
Description:

- The parser is composed of multiple classes and interfaces that collaborate to provide math evaluation services. This division of labour is required in order to obtain a clean design. However, the parser users don't care about internal design, and thus need a user-centered interface. The façade offers this perspective.

Consequences:

- Improve the parser ease of use by hiding implementation details.
- Reduce the coupling between the clients and the library by minimizing the number of visible classes.

Collaborators:



- Client

The client is the user of the library. For example, your application code. The only class you need to know about is **MTParser**.

- **MTParser**

This is the façade class. It offers the services of a registrar and a compiler, all in one interface. In addition, it defines helper methods to simplify the execution of frequent tasks.

- **MTParserRegistrar** and **MTParserCompiler**

The parser is composed of multiple objects (for instance, a compiler and a registrar) in order to not put the entire burden in the same object, which would be too huge. The **MTParserRegistrar** and the **MTParserCompiler** objects each handles a part of the math evaluation task.

Prototype

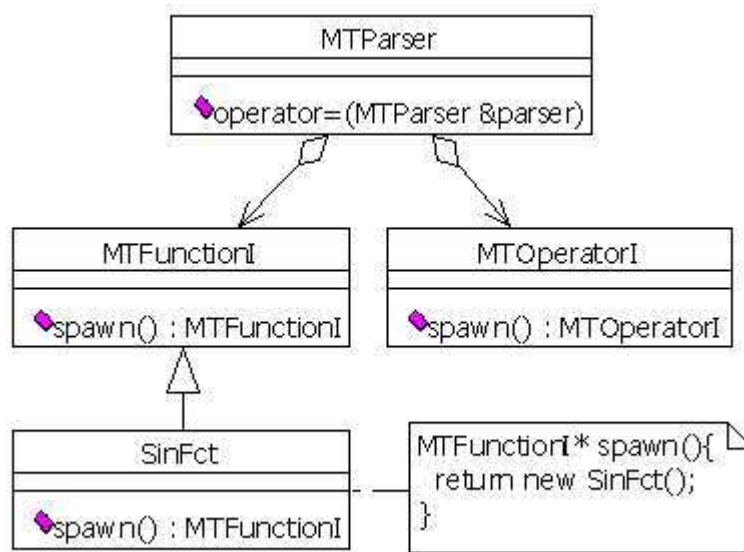
Description:

- Custom functions and operators can be defined at run-time, so the parser doesn't know these objects. In this case, how can the parser duplicate these objects when it needs to clone itself? A prototype is an object that supports cloning. Consequently, all function and operator objects are prototypes that can clone themselves. The **spawn** method is used to ask an object to duplicate itself.

Consequences:

- Allows run-time item definitions and creations (functions, operators...).
- There is no coupling between the items and the parser although the parser needs to create item instances. This is very different from a factory-based creation pattern where the factory needs to know about concrete classes.
- The item classes handle the creation task complexity. In fact, an item needs to know how to duplicate itself. This is a small overhead when creating a new item class, but this avoids modifying the parser code.

Collaborators:



- **MTParser**

When initialising a parser from an existing parser object (using the **=** operator C++ or the **Copy** method in

the COM version), all defined items need to be duplicated. Then, the parser asks each item to build a new item instance. The new instance is totally independent, but has the same state as the original item.

- **MTFunctionI** and **MTOperatorI**

These are the item interfaces. The **spawn** method is used to ask for a duplicated object.

- **SinFct** and any other concrete item classes

These classes know how to clone themselves. The creation may consist of only a class instantiation, or of many operations to duplicate the object states. The good thing is that this logic is encapsulated in the concrete item classes, and no other knows about it.

Abstract Factory

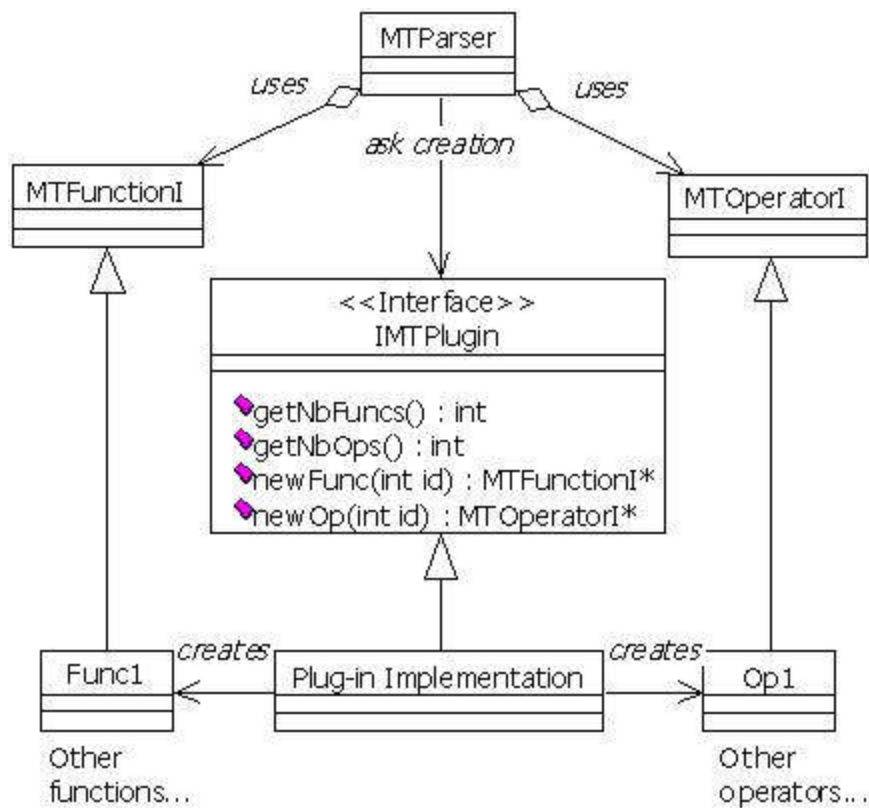
Description:

- The abstract factory pattern is about defining a factory interface and using it to obtain objects. The difference with the factory method pattern is that the concrete factory class isn't known until run-time. This can be transposed to the plug-in context. We want to be able to load unknown items (i.e., functions, operators, and constants) from an unknown DLL. The abstract factory interface corresponds to the COM plug-in IDL, and it has methods that return new item objects. The item classes can be implemented using native C++ classes. Once the parser has an object of each item, the plug-in is no more needed since the prototype pattern is used to duplicate existing items. This greatly simplifies the plug-in elaboration.

Consequences:

- Zero performance penalty when calling an item defined in a plug-in since this is a native C++ class (i.e., there is no COM call here, the object is accessed directly). COM calls are only made during the initialisation step, when defining the plug-in items.
- Because they are C++ classes, items can use the C++ parser services like custom compilation. There is no special interoperability consideration that would occur if a mix of C++ and COM objects were used.
- Plug-ins are simple to create since their COM interface is minimal. Only a few factory-related methods are needed. No COM interface for functions and operators is required.
- Plug-in maintenance is eased since the factory interface is pretty stable. However, item interfaces often change, but it is easier to change a C++ class' interface than a COM interface.

Collaborators:



- **IMTPlugin**

This is the abstract factory interface. It has methods to create new function and operator objects. Once all items are created, the plug-in can be unloaded.

- Plug-in implementation

This is the concrete plug-in class. It knows all of the plug-in function and operator classes, and thus can create new objects and return them to the parser.

- **MTParser**

The parser uses function and operator objects through the **MTFunctionI** and the **MTOperatorI** interfaces. It communicates with the plug-in using the **IMTPlugin** interface. When a plug-in is loaded, the parser asks the factory for one object of each item. After that, the parser uses the item objects directly.

- **MTFunctionI** and **MTOperatorI**

The C++ interfaces defining the services offered by the functions and operators.

- **Func1** and **Op1**

These are the functions and operators defined in the plug-in. Item code is dynamically loaded in memory by the operating system, using the DLL mechanism. Once created, the parser object only needs memory pointers that are obtained through the plug-in interface.

State

Description:

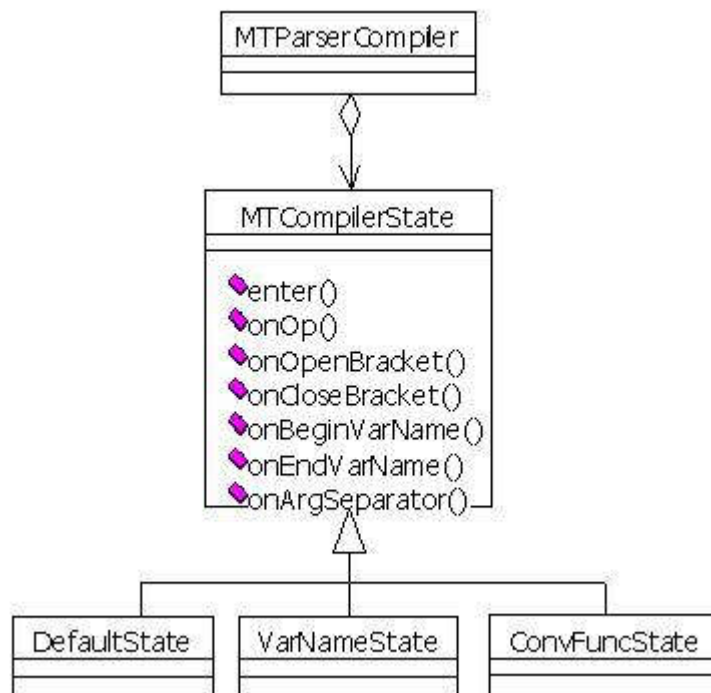
- The compiler interprets characters differently depending on its current state. For example, while parsing a conversion function argument, operator characters don't mean anything. The state pattern greatly simplifies the compiler code by avoiding duplication and having to handle special cases with multiple complex "if"

statements. A solution is to put the code related to each state into a separate class. This way, the compiler can be divided into a general parser and multiple states.

Consequences:

- New states can be added without impacting the compiler code. This means that new compiler features can be added quite easily. A new feature is implemented by creating a new state class instead of adding code to an already huge parsing method.
- The parsing and the compiling task can be separated. This allows a general parser to identify tokens, and delegate the interpretation task to the current compiler state.
- There is no big monolithic compiler class. Instead, there are several small classes, and a general compiler class that provides common services to the state classes.

Collaborators:



• **MTParserCompiler**

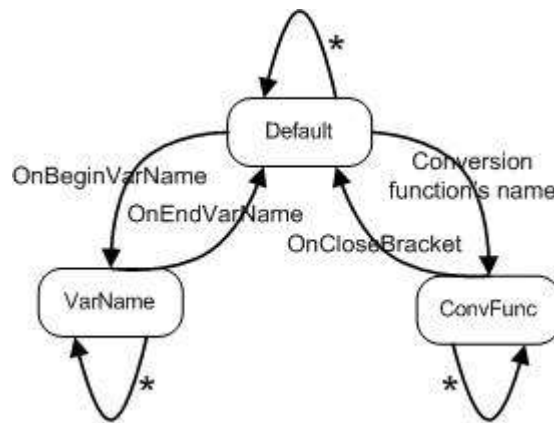
This is the expression compiler. It offers parsing (tokenizer) and general stack management services. When a token is identified (for example, an operator name), it calls the current compiler state object.

• **MTCompilerState**

This is an interface to which all states must conform. Each method corresponds to an identified token event. For example, when an operator name is detected, the **onOp** method is called. It is up to the state to decide what to do with each event. It also decides when to change the current state.

• Concrete states

There are currently three states. The following transition diagram shows what triggers each state.



The parser design

The first diagram presents the composition of a math expression. The **MTParser** class is a façade that hides two more complex classes that are a registrar and a compiler. The registrar responsibilities are:

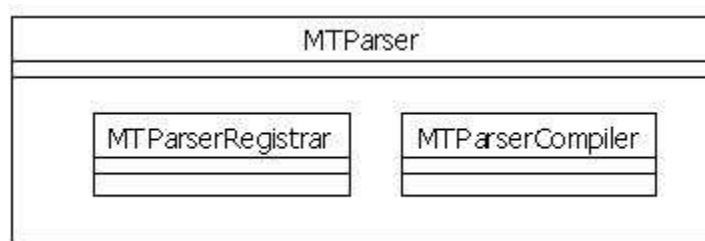
- Register valid expression items.
- Make sure that there is no item in conflict.
- Make sure that the whole "language" is consistent.
- Give access to the registered items.

The compiler responsibilities are:

- Take an expression in the infix notation, and transform it to a stack whose items can be evaluated sequentially.
- Validate the expression syntax.

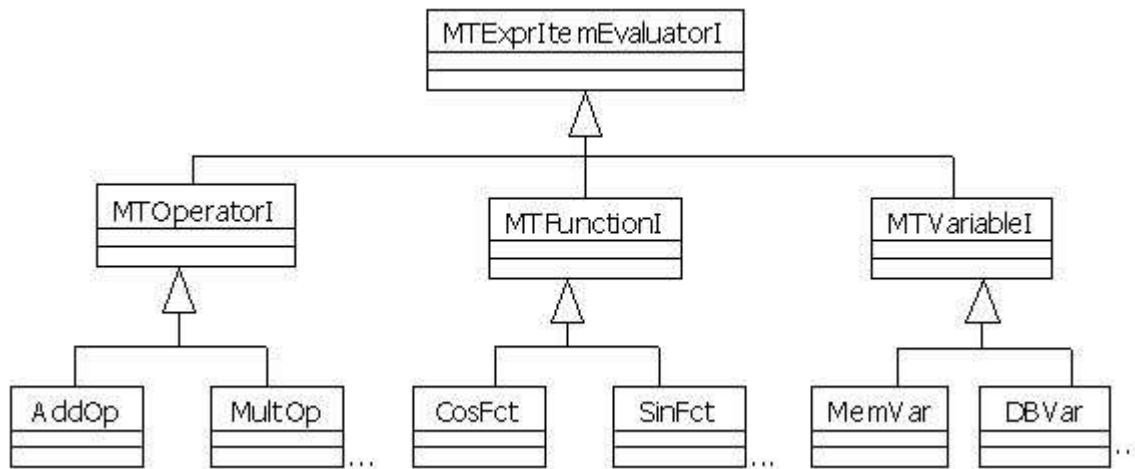
The separation of the parser in three objects actually reduces the complexity of each object, and will ease the handling of special registrars or compilers. For example, we can naturally imagine a compiler that would do certain kinds of optimizations that the user can choose to use or not.

There could have been another object called the evaluator. This object would be responsible to evaluate the expression result using the stack done by the compiler. The problem is that this would introduce one call indirection from the façade to the evaluator. Sadly, this indirection is too costly because the evaluate method is called repetitively, and thus each indirection time is multiplied thousand times.

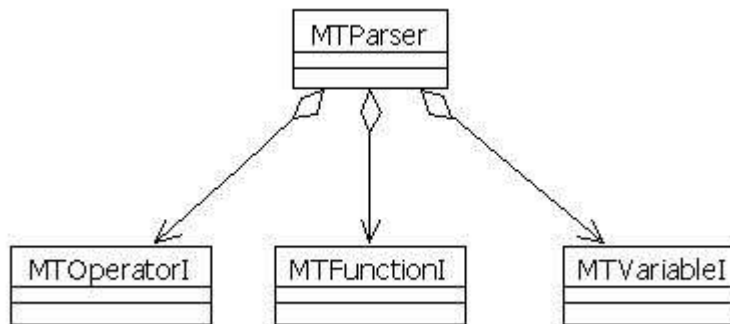


The following class diagram shows how operators, functions, and variables are abstracted. At the bottom, we see the concrete operator and function classes. New operators and functions can be added simply by implementing the **MTOperatorI** and the **MTFunctionI** interfaces. The difference between an operator and a function is that a function can have any number of parameters and an operator has a variable precedence (functions must have a greater precedence than operators in order for the parsing algorithm to work properly).

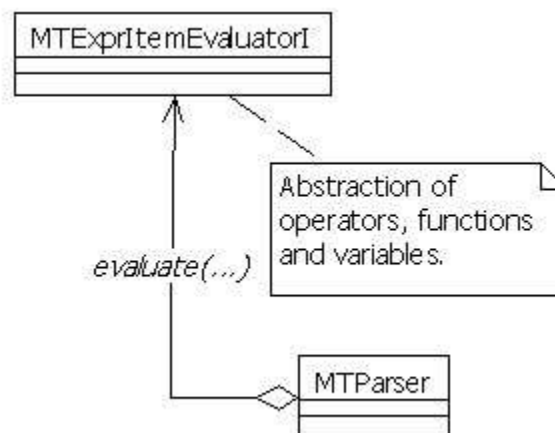
An alternate design could be to merge these two interfaces, but this could lead to bad function and operator implementations since operators are limited to two arguments and the precedence is fixed for functions. So in this case, I prefer to increase the design complexity to avoid coding error, by adding an interface. The **MTExprItemEvaluatorI** interface defines the generic methods required for the evaluation of an expression. It is the key to shield the math expression evaluator against changes and to keep the code clean.



The next class diagram presents the math parser dependencies (the **MTParser** class). We see that it must know whether the expression item is a function, an operator, or a variable, in order to handle each type specifically. This means that if a new item type is added, the parser will need to be modified. There is certainly room for improvement here, but this type of change may not be frequent enough to pay off for the possible loss of parsing speed.



Finally, the next class diagram presents the class relations during expression evaluations. There is no specific item interface (i.e., **MTOperatorI**, **MTFunctionI**...), and that means that the evaluation logic will not need to be changed unless the very most generic interface is changed (i.e., **MTExprItemEvaluatorI**).



Algorithms

The Parsing Algorithm

Basically, this math parser allows you to use the following items in the definition of your math expressions: number values, constants, variables, operators, and functions. Basic. The math expression is then preprocessed, and a stack is built (note that this is a stack and not a tree) to speed up evaluations. This way, the big processing is done only once at the parsing time. This is a very common and efficient algorithm to parse math expressions, so I will not give a lot of details here.

Here are the main ideas. To begin, you have two stacks: one temporary stack that will be used to store operators (namely the operator stack), and another stack that will contain the pre-processed expression that will be used later to do the evaluation (name it the second stack). The parsing is done from left to right. So, if you get a number, you put it immediately on the second stack. You do the same thing for all subsequent numbers. Now, if you get an operator (name it **curOp**), you do the following things:

```
If precedence(curOp) > precedence(lastOp) then
    push curOp on the operator stack
Else
    While precedence(curOp) <= precedence(lastOp)
        pop lastOp
        push lastOp on the second stack
    push curOp on the operator stack
EndIf
```

At the end, pop all remaining operators and push them on the second stack.

The goal of all this is to put high precedence operators at the bottom of the stack in order that they be evaluated first. Precedence here means the mathematical operator precedence. So, + and - have equal precedence, and * has greater precedence than +... Also, **lastOp** is the top element on the operator stack (in other words, it's the last operator pushed on the stack). At evaluation time, you pop items one by one and compute the result. The actual implementation does it recursively (see the code for more details). That's all! Of course, some code is needed to handle functions and brackets, but once you catch the basic idea, this is very straightforward.

In more technical terms, this algorithm is commonly called the shunting yard algorithm or simply an operator-precedence parsing algorithm. Its purpose is to transform an expression string using the infix notation to the reverse Polish form. The infix notation is what we, humans, commonly use to write mathematical expressions (for example, $x+2*2$), and the reverse Polish notation is what a computer can better understand because there is no need to worry about operator precedence and brackets. An example of the latter form is presented in the following section.

The Evaluation Algorithm

Given the expression 2^3+4*5 , applying the parsing algorithm gives the following stack (column 1):

#1	#2	#3	#4	#5
+	+	+	+	+
*	*	*	*	*
5	5	5	5	5
4	4	4	4	4
^	^	^	^	^
3	3	3	3	3
2	2	2	2	2

(Popped items are highlighted)

The evaluation algorithm pops the first item and evaluates it. In this case, the + operator is picked (#1). The addition operator needs two arguments, so it pops a first argument that is the * operator (#2). Again, the multiplication operator needs two arguments and pops a first argument that is 5 and a second argument that is 4 (#3). The *

operator can now be evaluated, and returns 20 that becomes the first argument for the + operator. Next, the addition operator pops a second argument that is ^ (#4). The ^ operator then pops two arguments, 3 and 2 (#5), and returns 8. Finally, the + operator returns 28, that is the sum of 8 and 20.

The pseudocode is:

```
double Evaluate()
    item = first item on the stack
    if item is a value
        return the value
    else
        return item->Evaluate()
    endif

double ItemXYZ::Evaluate(stack)
    return fctXYZ(stack->pop(),stack->pop(), ...)

double Pop()
    if the next item on the stack is a value
        return the value
    else
        return next item->Evaluate()
    endif
```

Again, this is the basic algorithm. Actually, it is used in the parsing step only, and a faster, iterative algorithm is used in the evaluation step.

Using the C++ Library and the COM Component

This section explains how to accomplish the most common tasks with the MTParser C++ library and the COM component. C++ code will be used to show how to use the C++ library, while, for the sake of diversity, VB6 and C# code will be used for the COM component.

Include the Parser in your Project

Before you begin evaluating math formulas, you have to integrate the parser with your application.

[C++]

Select the Unicode or the ANSI String Format

You have to choose! If you want to follow internationalization guidelines, you should use Unicode. You indicate your choice in the *"UnicodeANSIDefs.h"* file. What this file does is to define the right string manipulation functions depending on the string format you choose. For example, in the library code, instead of using **std:string** directly, **MTSTRING** is used. Then, if the Unicode version is used, **MTSTRING** will be defined as **std::wstring**. This is the trick that allows the library to use Unicode or ANSI strings (but not both!).

Other Compilation Flags

- **_MTPARSER_USE_PLUGIN**: if defined, the plug-in feature will be compiled.
- **_MTPARSER_USE_LOCALIZATION**: if defined, the localization feature will be compiled.

These flags are defined, by default, in the *MTParserPublic.h* file.

Compile the Library

If you don't have the *.lib* files, then you will have to compile the library by yourself. You'll be able to compile the library with Visual Studio 6, .NET 2003, and .NET 2005.

Note: Be sure to compile with the same run-time library version (for example, multithreaded DLL) as in your project.

The following list gives a short description of each project:

- **MTParserLib:** The core C++ parser library.
- **COM Library:** The COM parser library version. Allows multiple languages to use the parser functionalities. The COM library version is only a wrapper around the core library.
- **MTParserInfoFile:** The COM object to read the XML info file needed for localization purposes. It uses the .NET XML services, and is outside the core library because the .NET services are not available with VC6.
- **Plugins/MTParserPlugin:** The plug-in IDL interface to generate the TLB file.
- **Plugins/MTDatePlugin:** Date and time plug-in.
- **Plugins/MTNumAlgoPlugin:** Numerical algorithm plug-in.
- **Code Samples/C++ Library/C++ Client:** The main demo application using the C++ parser library version.
- **Code Samples/COM Library/C++ Client:** The C++ demo application using the COM parser library version.
- **Code Samples/COM Library/C# Client:** The C# demo application using the COM parser library version.
- **Code Samples/COM Library/VB Client:** The VB6 demo application using the COM parser library version.

If you only use the basic features, you only need to compile the MTParserLib project.

Visual Studio 6

Go to Project Settings|C/C++|Code generation, and then "Use run-time library". If you compile the library with a different run-time library than in your project, then you will get link errors stating that there are multiple symbol definitions.

With Visual Studio 6, you will be able to compile the following projects:

- *MTParserLib/MTParserLib.dsw*
- *Code Samples/C++ Library/C++ Client/MathExpr.dsw*

Visual Studio .NET 2003, 2005

Go to the project property pages, and then to C/C++|Code Generation, and then "Runtime library". Usually, you will use Multi-threaded Debug DLL in debug mode, and Multi-threaded DLL in release.

With Visual Studio .NET, you will be able to compile all projects except the VB6 example.

Include the Proper Header Files in your Project

The only header files you'll need to include are *MTParser.h* and *MTParserLocalizer.h*, if you use the localization features. The math parser class is **MTParser**:

```
#include <span class="code-string">"../MTParserLib/MTParser.h"</span>
#include <span class="code-string">"../MTParserLib/MTParserLocalizer.h"</span>
```

Note: I assume that the library is located in a directory beside your project. For example, if your project is in *c:\project\MyProject*, then the math parser library is in *c:\project\MyProject\MTParserLib*.

Link the Library to Your Project

There are four library versions, one for each mix of Debug/Release and ANSI/Unicode. You'll find the *.lib* files in the

lib directory under *MTParserLib*. You can add the proper library to your project link settings, or use the following pre-compiler command:

```
#ifdef _DEBUG
#ifdef _UNICODE
#pragma comment(lib, "../MTParserLib/lib/MTParserUd.lib")
#else
#pragma comment(lib, "../MTParserLib/lib/MTParserd.lib")
#endif
#else
#ifdef _UNICODE
#pragma comment(lib, "../MTParserLib/lib/MTParserU.lib")
#else
#pragma comment(lib, "../MTParserLib/lib/MTParser.lib")
#endif
#endif
```

The above code automatically links with the proper library version depending on your project settings.

I urge you to use the release version when benchmarking your project since it is really faster.

[VB6]

The only thing you have to do is to add the MTParser component in your project references. To do so in Visual Basic 6.0, go to Project|References|Browse and then browse for the *MTParserCOM.dll* file. The most common problem is to forget to register the COM file in your system. To do so manually, use the *regsvr32.exe* application located in your *Windows\System32* directory. Run "regsvr32 MTParserCOM.dll".

[C#]

The only thing you have to do is to add the MTParser component in your project references. To do so in Visual Studio .NET 2003, go to Project|Add Reference|Browse and then browse for the *MTParserCOM.dll* file. As stated in the VB6 section above, make sure to register the COM file. To use the parser namespace add the following line:

```
using MTPARSERCOMLib;
```

Create a Parser Object

You can create a new object and configure it from scratch, or you can copy an existing object configuration.

Initialize the Object Using the Default Configuration

This is the basic way of creating a new parser object. The default configuration consists of the default operators and functions, and uses the dot (.) as the decimal point character and the coma (,) as the function argument separator character.

[C++]

Call the empty constructor:

```
MTParser parser; // default configuration
```

[VB6]

Create a new object as usual:


```
Dim parser As New MTParser
```

[C#]

Create a new object as usual:

```
MTParser parser = new MTParser();
```

Initialize the Object Using Another Object Configuration

When you want to use multiple parser objects, it is more convenient to configure only one object and then create copies of it than having to configure them all. All the object state is duplicated, which includes: current math formula, variables, constants, custom functions, custom operators, and the syntax. This duplication is essential to ensure that the new object will continue to function properly although the original object is destroyed.

[C++]

There are two ways to configure a new object using another object configuration. The first way is to use the copy constructor, and the second is to call the assignment operator:

```
MTParser parserTemplate;
// Configure the template object...
parserTemplate.enableAutoVarDefinition(true);

// Using the copy-constructor to configure a new object
MTParser parser2(parserTemplate);

// Using the assignment operator to configure an object
MTParser parser3;
parser3 = parser2;
```

[VB6]

```
Dim parserTemplate As New MTParser
Dim parser As New MTParser
'Configure the template object...
parserTemplate.autoVarDefinitionEnabled = True

' Using the copy method to re-configure an object
parser.Copy parserTemplate
```

[C#]

```
MTParser parserTemplate = new MTParser();
MTParser parser = new MTParser();
// Configure the exprTemplate object...
parserTemplate.autoVarDefinitionEnabled = 1;

// Using the copy method to re-configure an object
parser.Copy(exprTemplate);
```

Evaluate Mathematical Expressions

This is the main purpose of this library! If you want to compute the result of a simple expression like "2+2", or of any

expression of arbitrary complexity and length, like "pi*min(x+y+sin(z)/2^3-40.9988*2, avg(y,x*10,3,5))", then this section shows you how it works.

Evaluate an Expression and Get the Result Immediately

This is the easiest and the more direct way of evaluating an expression.

[C++]

```
MTParser parser;  
MTDOUBLE result = parser.evaluate(_T("2+10*2"));
```

[VB6]

```
Dim parser As New MTParser  
Result = parser.Evaluate("2+10*2")
```

You handle errors the standard way by using the "On Error GoTo" statement.

[C#]

```
MTParser parser = new MTParser();  
double result = parser.evaluate("2+10*2");
```

Evaluate an Expression Multiple Times

If you want to evaluate an expression multiple times, with different variable values, the performance will be increased if you first compile the expression and then evaluate the expression. The following code evaluates 1000 times the same expression:

[C++]

There are two ways, one using a loop, and a second using a batch evaluate. In C++, both methods give the same performance but the second method is better when you already have a bunch of variable values to process. The reason is that variables' values will automatically be pulled from a vector.

The following code evaluates an expression multiple times in a loop:

```
MTDOUBLE x;  
MTParser parser;  
parser.defineVar(_T("x"), &x);  
parser.compile(_T("x+2*sin(x)"));  
  
for( int t=0; t < 1000; t++ )  
{  
    x = t; // new variable value  
    MTDOUBLE result = parser.evaluate();  
}
```

The following code evaluates an expression multiple times in one parser call:

```
unsigned int nbEvals = 1000;  
  
// Create variable object
```

```

MTDoubleVector *pX = new MTDoubleVector(_T("x"));

// Defines variables in the parser. pX is now owned by the parser.
MTParser parser;
parser.defineVar(pX);

// Allocate memory for variable values and results
MTDOUBLE *pXVector = new MTDOUBLE[nbEvals];
MTDOUBLE *pResults = new MTDOUBLE[nbEvals];

// Add your code to fill the value vector...

// Assign variable values
pX->setValues(pXVector, nbEvals);

// Compile the expression only once
parser.compile(_T("x+2*sin(x)"));

// Evaluate the expression for all variable value in one function call
parser.evaluateBatch(nbEvals, pResults);

```

[VB6]

The following code evaluates an expression multiple times in a loop:

```

Dim x As New MTDouble
Dim parser As New MTParser
x.Create "x", 1
parser.DefineVar x
parser.Compile "x+y*sin(z)"

For i = 1 To 1000
    x.Value = i ' new variable value
    result = parser.EvaluateCompiled()
Next i

```

The following code evaluates an expression multiple times but only does one function call. Note the special VB6 functions **setValueVectorVB6** and **evaluateCompiledBatchVB6**. Two versions of these methods are needed to accommodate VB6 and .Net.

```

Dim parser As New MTParser

Dim x As New MTDoubleVector
Dim y As New MTDoubleVector
Dim z As New MTDoubleVector

x.Create ("x")
y.Create ("y")
z.Create ("z")

parser.defineVar x
parser.defineVar y
parser.defineVar z

' Compile the expression only once
parser.compile "x+2*sin(x)"

' Generate random variable values...
Dim nbEvals As Long
nbEvals = 800000

Dim xval() As Double

```

```

ReDim xval(nbEvals) As Double

' Add your code to fill the value vector...

' Set values...
x.SetValueVectorVB6 xval

' this will contain all the results after evaluations
Dim results() As Double
ReDim results(nbEvals) As Double

' Evaluate the expression for all
' variable value in one function call
parser.evaluateCompiledBatchVB6 nbEvals, results

```

[C#]

Like in C++, there are two ways: one using a loop, and a second using a batch evaluate. But unlike in C++, the second method is really faster (as much as 40 times faster!) because COM calls are very slow.

The following code evaluates an expression multiple times in a loop:

```

int nbEvals = 1000;
MTDouble x = new MTDouble();
x.create("x", 0.0);

MTParser parser = new MTParser();
parser.defineVar(x as IMTVariable);
parser.compile("x+2*sin(x)");

for( int t=0; t < 1000; t++ )
{
    x.value = t; // new variable value
    double result = parser.evaluateCompiled();
}

```

The following code evaluates an expression multiple times but only does one function call:

```

MTDoubleVector x = new MTDoubleVector();
x.create("x");

MTParser parser = new MTParser();
parser.defineVar(x as IMTVariable);

// Allocate memory for variable values and results
double[] xval = new double[nbEvals];
double[] results = new double[nbEvals];

// Add your code to fill the value vector...

// Assign variable values
x.SetValueVector(xval);

// Compile the expression only once
parser.compile("x+2*sin(x)");
// Evaluate the expression for all
// variable value in one function call
parser.evaluateCompiledBatch(nbEvals, results);

```

Manage Variables

A variable is a special symbol that is replaced by a value when evaluated, but unlike a constant, its value can be changed. It can be used to make parameterized formulas like $3t+10$, where t is the time variable.

Define a Variable

Before you can use a variable in an expression, unless you use the automatic variable definition feature, you have to define it. Otherwise, the parser will not be able to recognize the variable and an exception will be thrown.

[C++]

The principle is to link a variable name in the parser to a real variable value in your program. In C++, this link is established through the use of a variable pointer.

The following code example defines a variable named **"x"** and links its value to the variable **"myVar"** in your program:

```
MTDOUBLE myVar;  
MTParser parser;  
parser.defineVar(_T("x"), &myVar);
```

When evaluating an expression containing a variable defined this way, the value is obtained automatically through the variable pointer. This method avoids you to have to set the variable value manually by calling a method like **"setVarValue"**. Instead, to change a variable value, you simply assign a new value to the C++ variable like:

```
myVar = 10;
```

[VB6]

The following code defines a variable named **"x"**.

```
Dim x As New MTDouble  
Dim parser As New MTParser  
x.Create "x", 1  
parser.defineVar x
```

The **Create** method initialises the variable object with a name and a default value. To change or get the variable value, use the **Value** property like this:

```
x.Value = 10 ' set the variable value  
val = x.Value ' get the variable value
```

[C#]

The following code defines a variable named **"x"**.

```
MTDouble x = new MTDouble();  
MTParser parser = new MTParser();  
x.create("x", 0);  
parser.defineVar(x as IMTVariable);
```

The **create** method initialises the variable object with a name and a default value. To change or get the variable value, use the **value** property like this:

```
x.value = 10 // set the variable value  
double val = x.value // get the variable value
```

Define Only the Used Variables

When the number of possible variables is very huge, defining them all up front, just in case only some were used in one expression, is not very optimised. Instead, you can ask the parser which variables are used and were not already defined. This lets you define only the needed variables. By default, the parser throws an exception if it encounters an undefined variable. To change this behaviour, you have to enable the automatic variable definition feature.

[C++]

The easiest way is letting the parser define the variables using default values, and then you redefine them with the desired variable objects. The following code shows how it works:

```

MTParser parser;
parser.enableAutoVarDefinition(true);
parser.compile(_T("x+y+z"));

double *pVars = new double[parser.getNbUsedVars()];

for( unsigned int t=0; t < parser.getNbUsedVars(); t++ )
{
    parser.redefineVar(parser.getUsedVar(t).c_str(), &pVars[t]);
}

```

First, you enable the automatic variable definition feature by calling the `enableAutoVarDefinition` method, and then you compile an expression. Second, you loop through each used variable, and redefine them pointing to valid memory locations. You could also redefine variables with new `MTVariableI` objects.

The factory design pattern fits perfectly here. It could be used to obtain variable objects or memory pointers associated with each variable. In fact, this design pattern is already implemented and the next section describes how it works.

If you provide a variable factory, it will be called when a variable needs to be defined. To create a variable factory, you have to implement the `MTVariableFactoryI` interface. Its only responsibility is to create new variable objects.

The following code shows how to enable the automatic variable definition feature and to set a variable factory:

```

class MyVariable : public MTVariableI
{
public:
    virtual const MTCHAR* getSymbol(){ return _T("symbol"); }
    virtual MTDDOUBLE evaluate(unsigned int nbArgs,
        const MTDDOUBLE *pArg){ return 1.32213; }
    virtual MTVariableI* spawn()
        throw(MTParserException){ return new MyVariable(); }
};

class MyVarFactory : public MTVariableFactoryI
{
public:
    virtual MTVariableI* create(const MTCHAR *symbol)
    {
        return new MyVariable();
    }

    virtual MTVariableFactoryI* spawn(){ return new MyVarFactory (); }
};

parser.enableAutoVarDefinition(true, new MyVarFactory());

```

The **"true"** parameter is to enable the feature, and the second parameter is the variable factory object. When the parser encounters an undefined variable, it will call the factory **create** method with the variable symbol.

[VB6]

The following code compiles an expression without defining variables beforehand, and then redefines each used variable:

```
Dim parser As New MTParser
parser.autoVarDefinitionEnabled = 1
parser.compile "x+y+z"
Dim vars() As MTDoube
ReDim vars(0 To parser.getNbUsedVars() - 1) As MTDoube

For t = 0 To parser.getNbUsedVars - 1
    Dim symbol As String
    symbol = parser.getUsedVar(t)
    Set vars(t) = New MTDoube
    vars(t).Create symbol, 0

    parser.redefineVar vars(t)
Next t
```

[C#]

The following code compiles an expression without defining variables beforehand, and then redefines each used variable:

```
MTParser parser = new MTParser();
parser.autoVarDefinitionEnabled = 1;
parser.compile("x+y+z");

MTDoube[] vars = new MTDoube[parser.getNbUsedVars()];

for(int t=0; t<parser.getNbUsedVars(); t++)
{
    vars[t] = new MTDoube();
    vars[t].create(parser.getUsedVar(t), t);
    parser.redefineVar(vars[t] as IMTVariable);
}
```

Extend the Parser

The parser language can be extended by defining custom constants, functions, and operators.

Define a Constant

A constant is a special symbol that is replaced by a value when evaluated. Using constants instead of values helps to make math formulas clearer, especially for well-known values. The parser can also do optimizations since it knows that these values are constants. Examples of constants are pi (3.14159...) and e (2.71...).

To define a constant, use the **defineConst** method. The first parameter is the constant name, and the second is the constant value.

[C++]

```
MTParser parser;
parser.defineConst(_T("pi"), 3.14159265359);
```

[VB6]

```
Dim parser As New MTParser
parser.defineConst "pi", 3.14159265359
```

[C#]

```
MTParser parser = new MTParser();
parser.defineConst("pi", 3.14159265359);
```

Define a Custom Function or Operator

C++ Library and COM using .Net Only

Operators and functions have been completely abstracted so that you can add your own at runtime and without having to modify the math parser code. This generalization simplifies the math parser a lot by eliminating all **switch cases** to select operators and functions. So, the code is pretty.

[C++]

The following code is all that is needed to define the summation function:

```
class SumFct : public MTFunctionI
{
    virtual MTSTRING getSymbol(){return _T("sum"); }

    virtual MTSTRING getHelpString(){ return _T("sum(v1,v2,v3,...)"); }
    virtual MTSTRING getDescription()
    { return _T("Return the sum of a set of values"); }
    virtual int getNbArgs(){ return c_NbArgUndefined; }
    virtual MTDDOUBLE evaluate(unsigned int nbArgs, const MTDDOUBLE *pArg)
    {
        MTDDOUBLE val = 0;

        for( unsigned int t=0; t < nbArgs; t++ )
        {
            val += pArg[t];
        }
        return val;
    }

    virtual MTFunctionI* spawn(){ return new SumFct(); }
};
```

The **getSymbol** method returns the function name as found in math expressions. The **getHelpString** method gives the proper syntax, and the **getDescription** method provides a short description of the function. The **getNbArgs** method returns the number of arguments needed by this function. The **c_NbArgUndefined** constant indicates that this function takes an undefined number of arguments. And finally, the **Evaluate** method computes the summation and returns the result. There is also a special method named **spawn** that returns a new **SumFct** object. The purpose of this method is to allow the duplication of parser configuration.

After creating your custom operator and function classes, you can tell the parser to use them using the following code:

```
try
{
    parser.defineOp(new MyOp());
    ...
    parser.defineFunc(new MyFct());
    ...
}
catch( MTParserException &e ){}
```

To define an operator, you call the **defineOp** method with your custom operator object. The **defineFunc** method does the same for custom functions. In the above example, the parser will automatically free the object memory at its destruction, so you don't need to keep a pointer on each created object and to delete it by yourself.

Operator and Function Overloading

This feature allows you to define multiple functions with the same name but with different number of arguments. For example, the Random function can take zero or two arguments:

```
// Random with zero argument
class RandFct : public MTFunctorI
{
    virtual MTSTRING getSymbol(){return _T("rand"); }

    virtual MTSTRING getHelpString(){ return _T("rand()"); }
    virtual MTSTRING getDescription()
        { return _T("Random value between 0 and 1"); }
    virtual int getNbArgs(){ return 0; }

    virtual bool isConstant(){ return false; }

    virtual MTDDOUBLE evaluate(unsigned int nbArgs, const MTDDOUBLE *pArg)
    {
        return rand()/((double)RAND_MAX);
    }

    virtual MTFunctorI* spawn(){ return new RandFct(); }
};
```

```
// Random with two arguments
class RandMinMaxFct : public MTFunctorI
{
    virtual MTSTRING getSymbol(){return _T("rand"); }

    virtual MTSTRING getHelpString(){ return _T("rand(min, max)"); }
    virtual MTSTRING getDescription()
        { return _T("Random value between min and max"); }
    virtual int getNbArgs(){ return 2; }

    virtual bool isConstant(){ return false; }

    virtual MTDDOUBLE evaluate(unsigned int nbArgs, const MTDDOUBLE *pArg)
    {
        return pArg[0]+(rand()/((double)RAND_MAX)*(pArg[1]-pArg[0]));
    }
};
```

```
}  
  
virtual MTFunctorI* spawn(){ return new RandMinMaxFct(); }  
};
```

The first Random function returns a value between 0 and 1. The second, more specialized function, allows the user to specify a range for the random value.

Another function overloading use is for performance optimization. Examples are the **min** and **max** functions. These functions can take an undefined number of arguments, but most of the time, only two or three arguments are used. So, three function prototypes are defined: two arguments, three arguments, and an undefined number of arguments. Functions with a defined number of arguments are generally faster since they just handle one predefined case; so the code can be optimized for this case (i.e., no loop).

[C#]

When you can't (or don't have the courage) to develop your function in c++ or in a plug-in, your other option is to implement the IMTFunction COM interface. The performance will be about 30% less in average. But I agree with you, this is really much easier!

Following is the code for a sum function taking an undefined number of arguments. As you see, multiple **evaluateX** methods can be implemented to provide better performance. If your function always takes 2 parameters, you only need to implement the **evaluate2** method. The special -1 value returned by the **getNbArgs** method indicates an undefined number of arguments.

```
public class MySumFunction : IMTFunction  
{  
    public double evaluate0()  
    {  
        throw new Exception("The method or operation is not implemented.");  
    }  
  
    public double evaluate1(double arg)  
    {  
        return arg;  
    }  
  
    public double evaluate2(double arg, double arg2)  
    {  
        return arg + arg2;  
    }  
  
    public double evaluate3(double arg, double arg2, double arg3)  
    {  
        return arg + arg2 + arg3;  
    }  
  
    public double evaluate(Array pArgs)  
    {  
        double sum = 0;  
        for (int t = 0; t < pArgs.Length; t++)  
        {  
            sum += (double)pArgs.GetValue(t);  
        }  
  
        return sum;  
    }  
  
    public string getDescription()  
    {  
        return "Computes the sum of many values";  
    }  
}
```

```

    }

    public string getHelpString()
    {
        return "slowsun(x,y,z,...)";
    }

    public int getNbArgs()
    {
        return -1;
    }

    public string getSymbol()
    {
        return "slowsun";
    }
}

```

The following code define your function:

```

MTParser parser = new MTParser();
parser.defineFunc(new MySumFunction());

```

Define a Macro Function

A macro function is helpful to simplify the writing of frequently used expressions. For example, if you often use the Euclidean distance defined as $\sqrt{x^2+y^2}$, then it may be a good idea to define a macro named "euc(x,y)".

To define a macro, use the `defineMacro` method. The first parameter is the macro prototype, the second is the macro function, and the third is a short description.

[C++]

```

MTParser parser;
parser.defineMacro(_T("euc(x,y)"), _T("sqrt(x^2+y^2)" ) ,
    _T("Euclidean distance"));

```

[VB6]

```

Dim parser As New MTParser
parser.defineMacro "euc(x,y) ", "sqrt(x^2+y^2)" , "Euclidean distance"

```

[C#]

```

MTParser parser = new MTParser();
parser.defineMacro( "euc(x,y)", "sqrt(x^2+y^2)", "Euclidean distance");

```

Load a Plug-in

Load functions, operators, and constants at run-time. The `loadPlugin` method takes a class identifier (CLSID) string. This string identifies a COM object, and should be provided to you by the plug-in developer. The CLSID (which can be stored as a string) should be retrieved from a file or the registry. This allows you to add and modify class identifiers after the application release, and thus achieves the goals of plug-ins.

[C++]

```
MTParser parser;  
parser.loadPlugin(_T("{4C639DCD-2043-42DC-9132-4B5C730855D6}"));
```

[VB6]

```
Dim parser As New MTParser  
parser.loadPlugin "{4C639DCD-2043-42DC-9132-4B5C730855D6}"
```

[C#]

```
MTParser parser = new MTParser();  
parser.loadPlugin("{4C639DCD-2043-42DC-9132-4B5C730855D6}");
```

Load All Available Plug-ins

The XML info file associated with each plug-in contains the plug-in CLSID. So, the parser can automatically look for info files and discover their CLSID. This is a very flexible way of handling plug-ins since you can add and remove plug-ins without modifying your program. The following code example loads all plug-ins located in the current directory:

[C++]

```
MTParser parser;  
MTSTRING directory = _T("./");  
MTSTRING pluginFileSearchPattern = _T("*.xml");  
parser.loadAllPlugins(directory.c_str(), pluginFileSearchPattern.c_str());
```

[VB6]

```
Dim parser As New MTParser  
parser.loadAllPlugins App.Path, "*.xml"
```

[C#]

```
MTParser parser = new MTParser();  
string directory = System.AppDomain.CurrentDomain.BaseDirectory;  
parser.loadAllPlugins(directory, "*.xml");
```

Localize the Parser

Many things need to be localized: error messages, item documentation (for example, function descriptions), and expression syntax (for example, decimal point and argument separator characters).

Initialize the Localizer

The localizer is a singleton (unique instance) object helping in multiple tasks of localization. Generally speaking, it contains localized strings for error messages and item documentation. So when you need a localized string, just ask the localizer.

Before using the localizer, you have to tell it which locale to use and where to find localized information. Locale strings follow a standard like the ISO two-letter language identifier. For example, English is "en" and French is "fr". Localized information is stored in XML files (the same files coming with plug-ins). An XML file contains one section for each locale. Next is an example of XML file:

```
<?xml version="1.0" encoding="utf-8" ?>
<LibraryInfo schema="2" type="static" data1="" data2="" version="3">
  <Resource LCID="en">
    <function id="sin" symbol="sin" args="x" argDescs="" description="Sine" />

    <function id="asin" symbol="asin" args="x"

      argDescs="" description="Arcsine" />
    <function id="sinh" symbol="sinh" args="x"

      argDescs="" description="Hyperbolic sine" />
    ...
    <operator id="+" symbol="+" args="x,y"

      description="Add two numbers" />
    <operator id="minus" symbol="-" args="x,y"

      description="Substract two numbers" />
    <operator id="unaryMinus" symbol="-" args="x"

      description="Unary minus" />
    ...
    <exception id="MTDEFEXCEP_SyntaxArgDecConflict"

      description="The argument separator character
        and the decimal point character
        are the same"/>
    <exception id="MTDEFEXCEP_SyntaxArgVarConflict"

      description="The argument separator character and one of
        the variable name delimiter
        characters are the same"/>
    <exception id="MTDEFEXCEP_SyntaxDecVarConflict"

      description="The decimal point character and one of the
        variable name delimiter
        characters are the same"/>
    ...
  </Resource>
  <Resource LCID="fr">
    <function id="sin" symbol="sin" args="x"

      argDescs="" description="Sinus" />
    <function id="asin" symbol="asin" args="x"

      argDescs="" description="Arc sinus" />
    <function id="sinh" symbol="sinh" args="x"

      argDescs="" description="Hyperbolique sinus" />
    ...
    <operator id="+" symbol="+" args="x,y"

      description="Additionne deux nombres" />
    <operator id="minus" symbol="-" args="x,y"

      description="Soustrait deux nombres" />
```

```

<operator id="unaryMinus" symbol="-" args="x"
    description="Soustraction unaire" />
...
<exception id="MTDEFEXCEP_SyntaxArgDecConflict"
    description="Les caractères de séparation d'arguments
        et de point décimal sont les mêmes"/>
<exception id="MTDEFEXCEP_SyntaxArgVarConflict"
    description="Les caractères de séparation d'arguments
        et de délimiteurs de noms de variables sont les mêmes"/>
<exception id="MTDEFEXCEP_SyntaxDecVarConflict"
    description="Les caractères de point décimal et de
        délimiteurs de noms de variables sont les mêmes"/>
...
</Resource>
</LibraryInfo>

```

The **LibraryInfo** node has four attributes. The **schema** attribute is a version number to validate the XML file format or schema. For example, in future releases, sections may be added or removed and the schema number will be incremented accordingly. The **type** attribute indicates whether the info file is associated with a plug-in. In this case, the **data1** attribute is set to the plug-in file name, and the **data2** attribute to the plug-in CLSID. This allows automatic plug-in loading by enumerating info files and retrieving CLSID.

Following are the resource sections. The **LCID** attribute is the locale string identifier. So when you ask for a certain locale, the localizer tries to find a resource section with the corresponding LCID. In a resource section, functions, operators, constants, variables, and exceptions are defined.

The following code sets the locale to use to French (fr) and loads all info files present in the current directory:

[C++]

```

MTParserLocalizer::getInstance()->setLocale(_T("fr"));
MTParserLocalizer::getInstance()->registerAllLibraries(_T("./"), _T("*.xml"));

```

[VB6]

```

Dim localizer As New MTParserLocalizer
localizer.setLocale "fr"
localizer.registerAllLibraries App.Path, "*.xml"

```

[C#]

```

string dir = System.AppDomain.CurrentDomain.BaseDirectory;
MTParserLocalizer localizer = new MTParserLocalizer();
localizer.setLocale("fr");
localizer.registerAllLibraries( dir, "*.xml");

```

Localize Error Messages

Error messages are declared in XML files, thus allowing you to easily add supported languages and fix translation mistakes. These info files are loaded at runtime. The following line is part of the English section of an XML file, and declares an exception:

```
<exception id="MTDEFEXCEP_OpAlreadyDefined"
    description="Operator already defined: '%itemName'"/>
```

- **id**: Unique exception string identifier. Using strings instead of numbers allows new exceptions to be defined (by plug-ins, for example) without wondering about used ID.
- **description**: Exception message with parameters. The parameters will be replaced by the actual exception values at runtime.

The following code uses the localizer to obtain a localized error message:

[C++]

```
MTSTRING getAllExceptionLocalizedString(const MTParserException &e)
{
    MTSTRING msg;

    for( unsigned int t=0; t<e.getNbDescs(); t++ )
    {
        MTSTRING desc;

        // Take the localized exception description if available
        try
        {
            desc =
                MTParserLocalizer::getInstance()->getExcep(
                    e.getException(t)->getData());
        }
        catch( MTParserException )
        {
            // description not available...so take the default english message
            desc = e.getDesc(t).c_str();
        }
        msg += desc;

        if( t != e.getNbDescs()-1 )
        {
            msg += _T("\r\n");
        }
    }

    return msg;
}
```

[VB6]

```
Private Function getLastExcepText(parser As MTParser) As String

    Dim Msg As String
    Dim e As New MTEXcepData
    Dim localizer As New MTParserLocalizer
    Dim desc As String

    Do
        parser.getLastExcep e

        If e.getID() <> "ok" Then
            desc = getLocalizedExcepText(e, localizer)
            If desc = "" Then
```

```

        ' Take the default description
        desc = e.getDescription()
    End If
    Msg = Msg + desc
    Msg = Msg + vbCrLf
End If
Loop Until e.getID() = "ok"

getLastExcepText = Msg
End Function

' Get the localized exception text. Return an empty string if not available
Private Function getLocalizedExcepText(data As MTEXcepData, _
    localizer As MTParserLocalizer) As String
    On Error GoTo unavailableDesc

    getLocalizedExcepText = localizer.getExcep(data)
Exit Function
unavailableDesc:
    getLocalizedExcepText = ""

End Function

```

[C#]

```

string getLastExcepText(MTParser parser)
{
    string msg = "";
    MTEXcepData e = new MTEXcepData();
    MTParserLocalizer loc = new MTParserLocalizer();

    do
    {
        parser.getLastExcep(e);
        if (e.getID() != "ok")
        {
            string desc = "";
            try
            {
                desc = loc.getExcep(e);
            }
            catch( Exception )
            {
                // No localized description available,
                // so take the default text
                desc = e.getDescription();
            }

            msg = msg + desc;
            msg = msg + Environment.NewLine ;
        }
    } while(e.getID() != "ok");

    return msg;
}

```

Localize Item Documentation

Functions, operators, constants, and variables all come with user documentation. This allows users to learn and remember what each item does and how to use it.

The following line is part of the English section of an XML file, and gives the documentation for a function:

```
<function id="randminmax" symbol="rand" args="min,max"
  argDescs="lower bound,upper bound"
  description="Random value between min and max" />
```

- **id**: Unique identifier since overloaded functions are allowed. For example, there are two random functions: one takes no argument and returns a number between 0 and 1, and the other takes a minimum and maximum value and returns a number between the two.
- **symbol**: Function symbol. Cannot be translated yet. So for now, the symbol needs to be the same in all languages.
- **args**: Function parameter name list separated by a coma.
- **argDescs**: Function parameter description list separated by a coma.
- **description**: Function description.

Localize the Parser Syntax

To adapt to international users, you can configure the syntax according to your user preferences.

The following code automatically uses the user syntax preferences registered in the Windows system:

[C++]

```
MTParser parser;
parser.useLocaleSettings();
```

[VB6]

```
Dim parser As New MTParser
parser.useLocaleSettings
```

[C#]

```
MTParser parser = new MTParser();
parser.useLocaleSettings();
```

You can also manually specify your syntax settings for the following items:

- Decimal point: x.yyyyy
- Function argument separator: f(x,y)
- Begin-end variable name separators: [myVariable]

[C++]

The following code reads the locale decimal point character and sets the parser syntax accordingly:

```
MTParser parser;
LCID lcid = GetThreadLocale(); // get the current locale id
MTCHAR buf[4];
if( GetLocaleInfo(lcid, LOCALE_SDECIMAL, buf, 4) )
// get the decimal point character
{
  MTSYNTAX syntax;
```

```

syntax = parser.getSyntax();

// make sure that the argument separator character is different
// from the decimal point character
syntax.argumentSeparator = ',';
if( buf[0] == ',' )
{
    syntax.argumentSeparator = '.';
}

syntax.decimalPoint = buf[0];
parser.setSyntax(syntax);
}

```

The **getSyntax()** method retrieves the current syntax. This allows you to set only the syntax elements that make sense to you and to use the default elements for the remaining. Next, the function argument separator character is set to the coma character and the decimal point is set to the dot character. In French, the function argument separator could have been set to ‘,’ and the decimal point character to ‘.’. The **setSyntax()** method actually registers the syntax.

[VB6]

```

Dim parser As New MTParser
Dim syntax As sMTSyntax
syntax = parser.getSyntax
syntax.decimalPoint = ASC(",")
syntax.argumentSeparator = ASC(";")
parser.setSyntax syntax

```

[C#]

```

MTParser parser = new MTParser();
sMTSyntax syntax = parser.getSyntax();
syntax.decimalPoint = ',';
syntax.argumentSeparator = ';';
parser.setSyntax(ref syntax);

```

A Little Localization Note

It is better to have some redundancies in the strings than to split sentences in little “reusable” words. For example, if you have these two error messages: “Operator name cannot be null” and “Function name cannot be null”, you could write things like:

```

//(IDS_xxx could be strings in your application resources)

std::wstring FormatNullError(int itemType)
{
    std::wstring msg;

    if itemType == Function then
        msg = IDS_FUNCTION
    else
        msg = IDS_OPERATOR

    msg += IDS_CANNOT_BE_NULL

    return msg;
}

```

This is so English-centric! First of all, in some languages (like French), you must know the noun genre to properly accord the verb. In the above example, you don't have this information, so the **IDS_CANNOT_BE_NULL** string cannot be translated accurately. Secondly, the word order may need to be modified. In French, the sentence is inverted and becomes: "Le nom de la fonction ne peut être null", that is the equivalent of "The name of the function cannot be null". It's a "La" for a function and a "L' " for an operator. Conclusion: you don't want to concatenate words at runtime. This is the translator job, and he has the tools to translate redundant words very efficiently and at low cost for you. Your job is to write complete and self-contained sentences.

Handle Errors

Special effort has been done to ease the error-handling task. Errors are reported uniformly throughout the API using the exception mechanism. No method returns strange error codes.

[C++]

There are two exception classes: **MTEException** and **MTParserException**. The former is a generic exception that contains only the exception description. With this exception, you don't get any exception parameters. The only advantage is that the default error description is easily accessible. The second exception class contains supplementary fields, allowing a localized error message to be produced.

The **throw(MTParserException)** declaration clause has been used everywhere a method could throw an exception. So when you see this code appended to a method prototype, be sure to use a **try-catch** statement when calling it.

The following code shows how to catch both exceptions:

```
try
{
    ...
}
catch( MTEException &e )
{
    // catch the simplified exception
    Display(e.m_description);
}
```

Or

```
try
{
    ...
}
catch( MTParserException &e )
{
    // catch the detailed exception
    Display(e.getException(0)->getData().description);
}
```

Actually, the exception mechanism supports chained exceptions. This is used to report complex exceptions that occur through multiple call layers. In this case, each layer can add contextual information to the exception. The result is that different levels of details are available if needed. The following code shows how to display all exceptions contained in an object:

```
try
{
    ...
}
catch( MTParserException &e )
```

```

{
    for( int t=0; t < e.size(); t++ )
    {
        Display(e.getException(t)->getData().description);
    }
}

```

The exception at index 0 occurred last, and consequently is the more general exception.

[VB6]

The description of methods that throw exceptions have been appended with the **"THROWS EXCEPTION"** statement. So when you see these words, be sure to use an **ON ERROR** statement when calling this method. The following code shows how to handle an error:

```

Private Sub MySub()

On Error GoTo errorHandler

...

Exit Sub
errorHandler:
    MsgBox "Error: " & Err.Description

End Sub

```

Like the C++ version, chained exceptions are supported. The following function retrieves all available exception descriptions:

```

Private Function getLastExcepText(parser As MTParser) As String

    Dim Msg As String
    Dim e As New MTEXcepData

    Do
        parser.getLastExcep e

        If e.getID() <> "ok" Then
            Msg = Msg + e.getDescription()
            Msg = Msg + vbCrLf
        End If
    Loop Until e.getID() = "ok"

    getLastExcepText = Msg
End Function

```

You simply pool the parser object until it returns the **"ok"** exception.

[C#]

The following code returns a message containing all exception texts:

```

string getLastExcepText(MTParser parser)
{
    string getLastExcepText = "";
    string Msg = "";
    MTEXcepData e = new MTEXcepData();
    do
    {

```

```

    parser.getLastExcep(e);
    if (e.getID() != "ok")
    {
        Msg = Msg + e.getDescription();
        Msg = Msg + Environment.NewLine;
    }
} while(e.getID() != "ok");
getLastExcepText = Msg;
return getLastExcepText;
}

```

You simply pool the parser object until it returns the "ok" exception.

Using COM Plug-ins

Plug-ins allow to load functions, operators, and constants at run-time. They need to be written in C++.

Create a Plug-in

Here are the steps to create a plug-in using Visual C++ 6.0:

1. Create a new ATL/COM project.
2. In the ClassView tab, create a new simple ATL object.
3. I suggest that you use the following ATL object properties:
 - Threading model = Free. This allows your plug-in to be used in any thread without the need of special marshalling code.
 - Check the "Free threaded marshaller" box. Same effect as the preceding setting.
 - Check the "Support ISupportErrorInfo" box. This allows your plug-in to throw "rich" exceptions with text and exception ID.
4. Implement the **IMTPlugIn** interface. After creating your ATL object, you go in the ClassView tab and right-click on the object class. Then, select "implement interface". Next, click "Add Typelib" and browse for the "MTPluginIDL.tlb" file of the MTPluginIDL project.

Implement a Plug-in

The plug-in interface is a simple factory. The only thing you have to do is to create objects and return them. You create functions and operators the standard way by implementing the **MTFunctionI** and the **MTOperatorI** interfaces.

You may want to look at the date plug-in implementation for more details. When using this design, the only code that needs to be changed is the initialization code that fills the constant, operator, and function arrays. This code looks like:

```

void CMTDatePlugin::initPlugin()
{
    // constants...
    addConst(_T("cname1", cval1);
    addConst(_T("cname2", cval2);
    ...

    // operators...
    m_ops.push_back(new myOp1());
    m_ops.push_back(new myOp2());
    ...
}

```

```
// functions...
m_funcs.push_back( new MyFct1() );
m_funcs.push_back( new MyFct2() );
...
}
```

The remaining of the plug-in code simply delegates calls to the right operators and functions.

Register a Plug-in

Before you can use a plug-in (and any COM object), you have to register it in the Windows system. There is a little system utility named "regsvr32.exe" that can do it for you. Simply run it in the command line using the following syntax: "regsvr32 myPluginFile.dll".

If you want to register a COM object programmatically, use the following C++ code:

```
BOOL RegisterCOM(const std::wstring &filename){
    HINSTANCE hLib = LoadLibrary(filename.c_str());
    HRESULT (*lpDllEntryPoint)();

    if (hLib < (HINSTANCE)HINSTANCE_ERROR)
    {
        return 0;
    }

    BOOL Success = 1;
    lpDllEntryPoint = (HRESULT (*)())GetProcAddress(hLib,
        "DllRegisterServer");
    if (lpDllEntryPoint != NULL)
    {
        HRESULT hr;
        hr = (*lpDllEntryPoint)();
        if (FAILED(hr))
        {
            Success = 0;
        }
    }
    else
    {
        Success = 0;
    }

    FreeLibrary(hLib);
    return Success;
}
```

The main idea is to call the **DllRegisterServer** function, which all COM objects have. This function will do all the remaining registration operations, like adding keys to the Windows registry.

Files to Distribute

Not all files are needed depending on which features you use. Here is a list of files and explanations when you need to distribute them with your application:

- *MTParserCOM.dll*: Needed when using the COM library version.
- *_MTParserPlugin.tlb*: Needed when using plug-ins. This is the type library file defining the plug-in interface.
- *MTDatePlugin.dll* and *MTDatePlugin.xml*: Needed when using date and time functions.
- *MTNumAlgoPlugin.dll* and *MTNumAlgoPlugin.xml*: Needed when using numerical algorithm functions: solve, derivate...

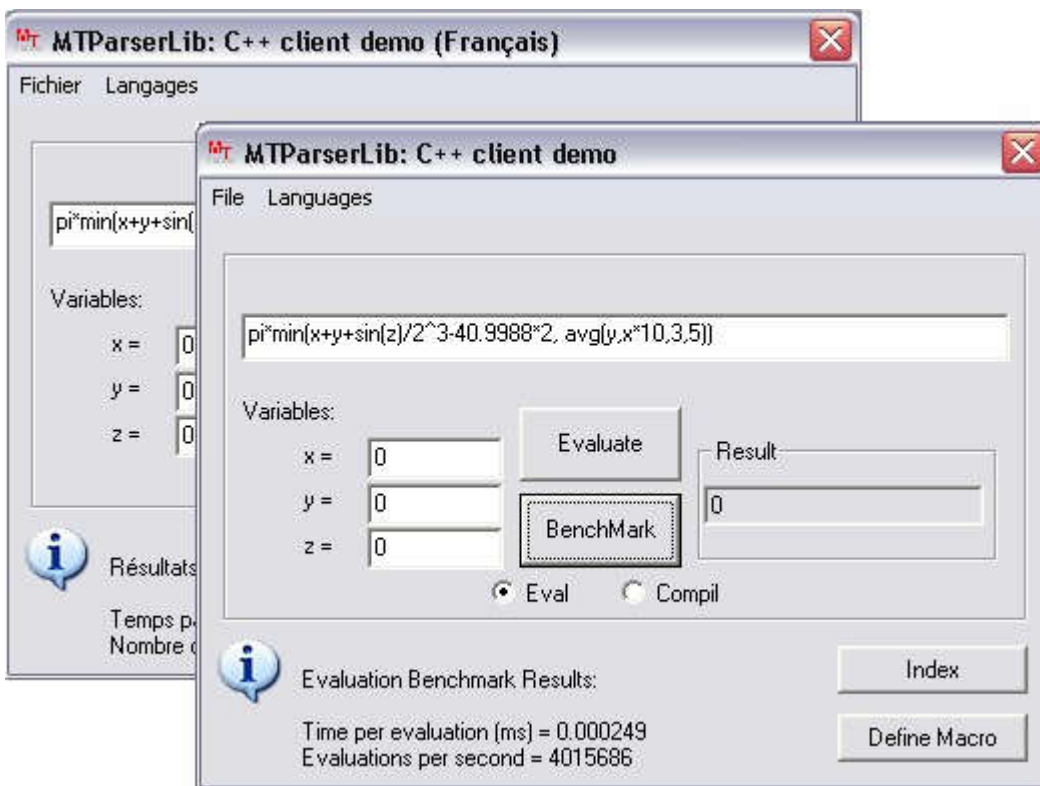
- *MTParserGen.xml*: Needed when using a localized parser. It contains text for exceptions, functions, and operators in different languages.
- *MTParserInfoFile.dll*: Needed when using the localization feature. This component reads XML info files.

So if you use the C++ library version without any extra feature, you don't need to distribute any file since the parser is statically linked with your application.

When deploying the COM object, you have to distribute the Visual Studio runtime DLL files. These files change depending on whether you use Unicode and your Visual Studio version. For example, when compiling using Unicode and Visual Studio .NET 2003, you'll need to distribute: *MFC71U.dll*, *MSVCP71.dll* and *MSVCR71.dll*.

A tool like [Dependency Walker](#) may help you determine needed DLLs and troubleshoot a problematic installation.

Demo Applications



There are four code samples:

- A C++ client using the C++ parser library. The main demo application featuring a localized user interface in English and French.
- A C++ client using the COM parser library.
- A VB client using the COM parser library.
- A C# client using the COM parser library.

All samples are equivalent, and demonstrate the same features. You can find them in the directory named *Code samples* under the *MTParser_src* main directory. In the *C++ library* sub-directory, you'll find a C++ application using the static MTParser C++ library. In the *COM library* sub-directory, you'll find three code samples using the MTParser COM library: one in C++, one in Visual Basic 6, and another in C#.

Other Math Parsers

There are many good math parsers out there. The following table presents eight math parsers and their

specifications. This table may be useful when you want to evaluate other parsers that fit your configuration needs.

	CioinaEval	EffOBJ	JFormula	JbcParser	JEP	MTParser	muParser	Ucalc	USPExpress
Native language	Delphi	C++	Java	Java	Java	C++	C++	C++	C++
Distribution format	DLL, So	DLL	JAR	JAR	JAR	.lib, COM	.lib, DLL	DLL	COM
Object oriented	N	N	Y	Y	Y	Y	Y	N	Y
Cross-platforms	Y	N	Y	Y	Y	N	Y	N	Y
Commercial license	\$	\$	\$	\$	\$	Ask	Ask	\$	\$
Personal license	\$	\$	\$	\$	GPL/\$	Free	Free	\$	\$
Source code	\$	N/A	\$	\$	GPL/\$	Free	Free	N/A	N/A

Conclusion

I really appreciate your comments, and I will continue to enhance this library regularly based on your feedback. The directives I follow when adding features are: to keep the library general enough to be useful for a maximum number of people, and "Keep It Simple Stupid". Don't hesitate to give me advises and to criticize my work. As long as you propose some solutions, you are welcome.

References

- [Grune, Dick. \(1990\). Parsing Techniques – A Practical Guide. Vrije Universiy, Amsterdam.](#) Complete and free online compiler book. Various algorithms are covered.
- [GSL: GNU Scientific Library.](#) A very complete and free collection of numerical algorithms in C. Wrappers could be developed to incorporate these algorithms with MTParser.
- [Niemann, Thomas. \(2000\). Operator-Precedence Parsing. ePaper Press.](#) Short and sweet paper explaining what is the operator-precedence parsing algorithm. This is the basic algorithm used by this component to parse mathematical expressions.
- [Norvell, Theodore. \(2001\). Parsing Expressions by Recursive Descent.](#) Good article explaining the shunting yard algorithm. This is just another way of naming the operator-precedence algorithm.
- [Press, William H. \(1992\). Numerical Recipes in C: The Art of Scientific Computing. Cambridge University Press.](#) Classic book on numerical methods. The online version is freely available, and should give you some plug-in ideas...!
- [Wikipedia. Reverse Polish Notation.](#) Very comprehensible definition and examples of the reverse Polish notation. This notation is used to represent mathematical expressions in a way that is more easily handled by a computer.

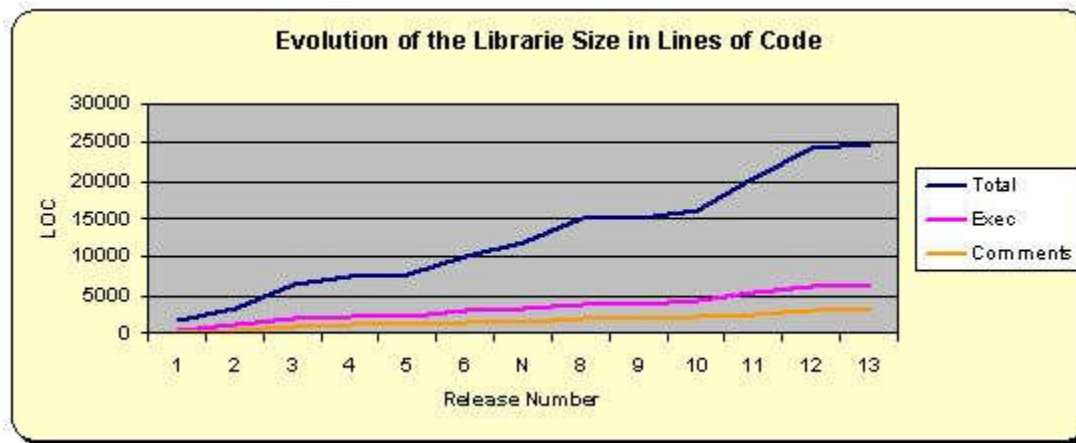
History

Now that the project is mature, there are some interesting metrics to be shown.

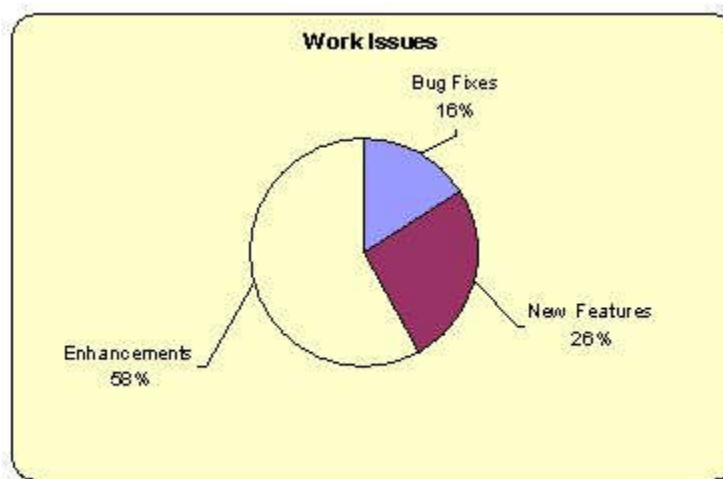
The following graph shows the evolution in lines of code (LOC) since the first release. The total number of LOC counts blank, commented, and executable lines. A C++ executable line ends with a semicolon (;).

Current Release Size (LOC):

- Total: 24896
- Executable: 6382
- Commented: 3222



Curious about how I spend my time on this project? The next graph tells us that more than half of the time I'm polishing existing things.



- March 10, 2008
 - Enhancement (Thanks to Rohallah Tavakoli): Method throw clauses revised to ease portability.
 - Enhancement: Function, constant, operator and variable names may contain the decimal point character. Since this character isn't used to split formula parts, it can be used in token names.
 - New Feature: VB6 methods for batch evaluate. VB6 needs different method signatures than .net in order to use safearray.
 - New Feature: User-defined function! Just implement the IMTFunction interface in your .net application!
 - Feature removed: Variable name delimiter characters. Useless feature and allowed strange syntax. Better to avoid variable name to contain special characters since it confuses the users.
 - Bug Fix (Thanks to Romy): The not operator now has the unary operator precedence. Test case added.
 - Bug Fix: Return copy of internal variable in redefineVar using the spawn method since registrar own them afterward.
- September 20, 2006
 - Enhancement (Thanks to udvranto): VS.NET 2005 ready.
 - Enhancement (Thanks to Jamie Riell): Operator precedences revisited. Now uses the C++ operator precedence values.
 - Enhancement (Thanks to ABuenger): Can now compile without the plug-in and localization features.
 - New Feature (Thanks to Bidoy): Unary addition operator added. Syntax like +2 is now valid.
 - New Feature (Thanks to Eldon Zacek and bastet69): Can undefine constants, functions, and operators
 - New Feature (Thanks to claude_kouakou): Can enumerate defined variables. **getNbDefinedVars** and **getVar** methods added.

- Feature Removed (Thanks to iberger): No more checking for out of memory exception. It is up to the application to handle the system exception. Historically, this feature was required when the **new** operator returned **null** instead of throwing an exception.
- Bug Fix: Expressions like "2,2" crashed the parser. Compiler method **getCurItemType** fixed.
- Bug Fix (Thanks to tux512): The **log10** function now uses the log10 math function instead of the log function!
- Bug Fix (Thanks to tux512): Expressions like 1+() crashed the parser. The () syntax is now invalid. Test case added.

- September 20, 2005

- Refactoring: Numerical algorithms moved in to a plug-in. These are specialized functions.
- Refactoring: Variable key removed. This was an unnecessary optimization and somehow ugly. Now only uses variables' names.
- Refactoring: Exception framework refactored. Exception identifiers are now strings instead of numbers and runtime configurable exception data. This allows new exceptions to be added without conflict. So, plug-ins can define their own exceptions and put localized text in their XML files.
- Enhancement - Robustness: Now catches out of memory conditions. When out of memory, an exception is thrown.
- New Feature: Exception message formatter to allow parameterized message to be formatted.
- Enhancement - Performance: In the registrar: Hash table for operators, multimap for functions, and map for constants. Now, searching functions are very performant. The end-result is better compilation time.
- New Feature: Now able to search for all plug-ins and load them automatically. The **loadAllPlugins** takes a search pattern to locate all info files and extract CLSID strings.
- New Feature: Batch evaluate. Allows an expression to be evaluated multiple times in the same call, and automatically pulls variable values at each evaluation.
- New feature: Localizer object to allow localized item information. XML files are used to store localized information. Uses the **MTParserLocalizer** object to obtain localized information about defined items. A COM object is used to parse XML files and use the .NET framework XML functions.
- New Feature: Redefined variable. This eases the use of the automatic variable definition feature since a variable factory is no more needed. You can compile an expression and redefine automatically defined variables using the proper variable type.
- New Feature: C# demo application. Thanks to Derek Price.

- June 20, 2005

- Refactoring: The **getSymbol** method is now in the **MTEExprItemEvaluatorI** interface. Before, there were differences between the symbol format of concrete classes (functions, operators, and variables) but now the format is the same for all.
- Enhancement - Memory (Thanks to Eamonn McGrath): Macros now use much less memory. Using the late initialization technique, macros are only created when actually used in the expression. New method added to the **MTFunctionI** interface to allow late initialization and avoid wasting resources if the function is not used.
- Enhancement - Error: Conversion functions allow no more missing closing bracket.
- Enhancement - Validation: Now validates functions, operators, and constants ID when calling the **getFunc**, **getOp**, and **getConst** methods. When the ID is out of range, an error is thrown instead of terminating the application completely.
- Enhancement - Performance: The VS.NET compiler provides a 15% speed improvement.
- New Feature: New function added, "**isFinite**", to allow handling evaluation errors like division by zero.
- New Feature (inspired by Eamonn McGrath): Macros' arguments can now be ordered, and does not have to follow the order of appearance of the macro function. Now you provide the macro prototype like: macro(arg1, arg2, arg3...) instead of only its name.
- Bug Fix: Problem with recursive functions like **trapezoid(2*x,x,trapezoid(2*x,x,0,5),0, 0.001)**. The arguments are now passed by value instead of by references.
- Bug Fix (Thanks to Martin Gamperl): Problem when using the auto-var definition feature with the

begin-end var delimiters. The var compiler state didn't respect the auto-var feature enabled state. Test case added.

- Bug Fix (Thanks to Visual Studio .NET!): There was a problem with void expressions (again). The program attempted to read an invalid memory space.

- April 23, 2005

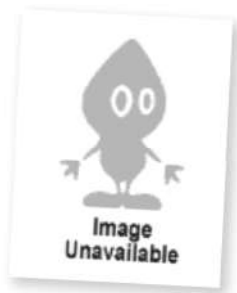
- Refactoring: The plug-in design now follows the abstract factory pattern. Developing a plug-in is now easier.
- Refactoring: The two exception types **MtDefException** and **MtParsingException** have been merged into **MtParserException**. Two exception types were not justified. The **MtException** still exists.
- Enhancement - Performance: Plug-in items are now as fast as native C++ items. Instead of calling a COM method to evaluate an item, an item instance is returned by the COM plug-in and used directly.
- Enhancement - Exception: Two new exceptions related to plug-ins: **e_MtDefExcep_PluginVersion** and **e_MtDefExcep_PluginNotFound**. The **e_MtDefExcep_ConvFuncSyntax** exception has been replaced by **e_MtParsingExcep_InvalidFuncSyntax** that is more generic.
- Enhancement - Compiler Performance: The compiling step is now faster. The main problem was the use of strings instead of simple character pointers.
- New Feature: Functions can have their own compiler. This allows a broad new range of functions to be defined, using special syntax. Conversion functions are no more special entities but now have a special compiler. This generalization simplifies the code, and opens the door for many kinds of new functions.
- New Feature: Numerical approximation functions. Derivative, integral, and solve functions have been implemented using the new custom compiler feature. These functions show what can be done, and are provided for demonstration purposes.
- Bug fix: Memory leak. Thanks to Dan Moulding's Visual Leak Detector at CodeProject!
- Bug fix (Thanks to mgampi!): Problem with the expression "(2,2)". Now validates that argument separators are used inside functions.

See the library source file for the full history.

License

This article, along with any associated source code and files, is licensed under [The Code Project Open License \(CPOL\)](#)

About the Author




Mathieu Jacques

Web Developer

Canada 🇨🇦

Software Engineer working at a fun and smart startup company

Comments and Discussions

 **720 messages** have been posted for this article Visit <https://www.codeproject.com/Articles/7335/An-extensible-math-expression-parser-with-plug-ins> to post and view comments on this article, or click [here](#) to get a print view with messages.

[Permalink](#) | [Advertise](#) | [Privacy](#) | [Terms of Use](#) | Mobile
Web01 | 2.8.180505.1 | Last Updated 13 Mar 2008

Article Copyright 2004 by Mathieu Jacques
Everything else Copyright © [CodeProject](#), 1999-2018