# ExprEval API

**Ali Khudiyev**

ExprEval

Evaluate your expression in C++

# Contents

# 1 Introduction

Expression-Evaluator (ExprEval) is a library written in C++ to evaluate strings as a mathematical expressions. If you have ever used "eval" in python you know it also evaluates strings, however, ExprEval is much more powerful than "eval"; it has many capabilities that "eval" lacks.

## 1.1 Priorities of operators

In a mathematical expression, all operators have their own priorities to indicate the order of calculation. It means, that the priorities tell us which operator has to be handled before than the other. Here is the list of operators and their corresponding priorities which have been defined and are usable in this project.

| **Operator** | **Priority** |
|---|---|
| $+$ | 9 |
| $-$ | 9 |
| $\times$ | 8 |
| / | 8 |
| $\wedge$ | 7 |
| mod | 7 |
| ln/log2/log | 6 |
| sin/cos/tan | 6 |
| sinh/cosh/tanh | 6 |
| arcsin/arccos/arctan | 6 |
| arcsinh/arccosh/arctanh | 6 |
| round/ceil/floor | 6 |
| abs/rand/gamma | 6 |
| and/or/not/xor/shiftl/shiftr | 6 |
| pi/e/T/F | 5 |
| () | 0 |

Table 1: Operator priority table

## 1.2 Expression vs Function

There are these two concepts known as "expressions" and "functions". In the library interface, these two concepts can be used for different purposes.

**Expression.** Expressions are to represent concrete and computable mathematical strings(expressions).

For example,

$$5 + 3 - 2$$
$$3^{2-1} * 1$$

**Function.** Functions are to represent abstract and non-computable mathematical strings(expressions) which means, they are not directly computable.

For example,

$$\begin{aligned} f(x) &= x^2 \\ add(x, y) &= x + y \\ A(r) &= \pi \cdot r^2 \end{aligned}$$

# 2 API specification

## 2.1 Namespaces

The global namespace is **ExprEval** and it has several sub-namespaces. User would usually use **Expression** and **Function** classes (maybe less often **Tokenizer** class as well) located in the global namespace. Here is the complete view of the global namespace:

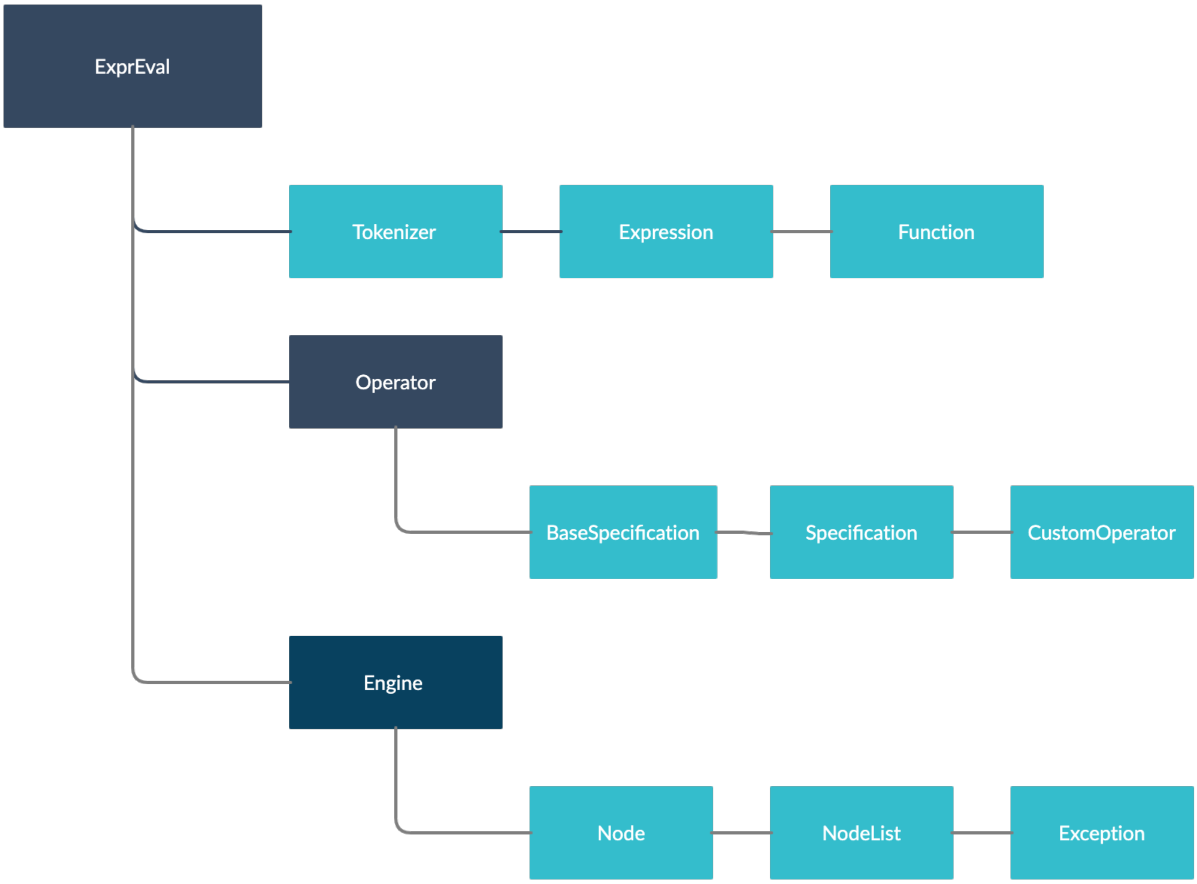


Figure 1: Namespace Tree

### 2.1.1 Tokenizer

You can use **ExprEval::Tokenizer** to tokenize any mathematical expression. To tokenize your string, you have to create an instance of **Tokenizer** class. Here are the couple of constructors:

```
Tokenizer();
Tokenizer(const std::string& expression);
```

If you have used the default constructor then you can set your expression with the help of setter:

```
void set(const std::string& expression);
```

After constructing the object, you can use the function below to get a single token each time:

```
Node get();
```

**Node** is a data structure which holds the data for a token. It has been defined as below:

```
struct Node{
    NodeType type;
    double number;
    std::string symbol;
    std::string expression;
};
```

```cpp
enum NodeType{
    Number = 0,
    Symbol,
    Expression ,
    Empty
};
```

**Note:** The program needs solid optimization as it is obvious from the Node structure.

### 2.1.2 Expression

You can use **ExprEval::Expression** to create and evaluate expressions. There are two constructors:

```cpp
Expression();
Expression(const std::string& expression);
```

If you have used the default constructor then you can set your expression with the help of setter:

```cpp
void set(const std::string& expression);
```

To evaluate your expression, there are several methods:

```cpp
double evaluate();
// To evaluate the set expression

double evaluate(const std::string& expression);
// To evaluate the (new) expression
```

### 2.1.3 Function

You can use **ExprEval::Function** to create and use your own functions. There are several constructors:

```cpp
Function();
Function(const std::initializer_list<std::string>& variables,
         const std::string& expression);
Function(const std::vectort<std::string>& variables,
         const std::string& expression);
```

There are also a couple of setters to initialize the constructed function:

```cpp
void set(const std::initializer_list<std::string>& variables,
         const std::string& expression);
void set(const std::vectort<std::string>& variables,
         const std::string& expression);
```

Functions can be computed with the help of operators declared as below:

```cpp
double operator(const std::initializer_list<double>& arguments);
double operator(const std::vectort<double>& arguments);
```

There are also a bunch of getters to be able to get [1]the true function symbol. There are some other overloaded getters to get the function symbol with some arguments which can be either numbers or another strings. The getter are shown below:

```cpp
std::string get();
std::string get(const std::initializer_list<double>& arguments);
std::string get(const std::vectort<double>& arguments);
std::string get(const std::initializer_list<std::string>& arguments);
std::string get(const std::vectort<std::string>& arguments);
```

[1] A function symbol is the symbol declared by the program implicitly. So, when you want to use your function in an expression you need to use that exact symbol for you function.

## 2.2 Specification of the built-in operators

All essential operators and functions in math have been implemented in the library. They have been implemented similarly to their mathematical definitions, therefore, they are very intuitive to use as expected.

| Operator | Arguments' positions | Priority | Description |
|---|---|---|---|
| $+$ | -1, 1 | 9 | $5 + 3 = 8$ |
| $-$ | -1, 1 | 9 | $8 - 2 = 6$ |
| $*$ | -1, 1 | 8 | $2 * 3 = 6$ |
| / | -1, 1 | 8 | $3/4 = 0.75$ |
| $\wedge$ | -1, 1 | 7 | $2^3 = 8$ |
| mod | -1, 1 | 7 | 7 mod 4 = 3 |
| ln/log2/log | 1 | 6 | log2(2) = 1 |
| sin/cos/tan | 1 | 6 | cos(0) = 1 |
| sinh/cosh/tanh | 1 | 6 | sinh(0) = 0 |
| arcsin/arccos/arctan | 1 | 6 | arccos(1) = 0 |
| arcsinh/arccosh/arctanh | 1 | 6 | arcsinh(0) = 0 |
| round/ceil/floor | 1 | 6 | round(3.8) = 4 |
| abs/rand/gamma | 1 | 6 | gamma(4) = 6 |
| not | 1 | 6 | not 1 = 0 |
| and/or/xor/shiftl/shiftr | -1, 1 | 6 | 1 shiftl 2 = 4 |
| pi/e/T/F | - | 5 | $T = 1$ |

Table 2: Specification of the built-in operators

The priorities of operators/functions cannot be changed by the user directly, however, they user has the ability to prioritize the operator(s) even with the least priority by using parentheses. For example, if you do not know in what order $2 \wedge 2 \wedge 3$ is going to be evaluated($(2^2)^3$ or $2^{(2^3)}$) then you can either use $(2 \wedge 2) \wedge 3$ or $2 \wedge (2 \wedge 3)$.

## 2.3 Errors and exception handling

There can be two main types of errors:

- User errors
- System errors

For both cases, the program has an exception handling methods to prevent the program crash and to be able to continue to work properly.

Here is the list of all error codes, their descriptions and handling methods.

| Error | Description | Handling | Exception |
|---|---|---|---|
| Symbol_Not_Found | Syntactic error | Throwing an exception | ExprEval::Engine::Exception |
| Expression_Logic | Semantic error | Throwing an exception | ExprEval::Engine::Exception |
| Memory_Allocation | Memory allocation failure | Immediate termination | - |
| File_Access | File access failure | Immediate termination | - |

Table 3: Errors and exception handling

A good way of using the library would look like this:

```
try{
    // Processing
} catch(const ExprEval::Engine::Exception& e){
    std::cerr << e.what() << std::endl;
    // Handling
}
```

### 2.3.1 User errors

There are a couple of errors which can be due to user behaviour:

1. Syntactic
2. Semantic

**Syntactic error.** This error appears due to an syntactically ill-formed expression. If the expression contains a symbol which has no mathematical definition then this is a syntactic error. Consider the example shown below:

$$5 + a$$

In this case, the error that the program throws would look like this:

**[ERROR]Symbol Not Found: node @ 2**

**Semantic error.** The cause of this error is due to an semantically ill-formed expression. Even if there is no syntactic error, there can still be a logical error. Consider the example shown below:

$$5 + +$$

In this case, the error that the program throws would look like this:

**[ERROR]Expression Logic: node @ 2**

### 2.3.2 System errors

# 3 Usage

1. Download the program from here.
2. Copy the **include** and **bin** folders to your project folder.
3. Include the **expreval.h** header file inside the **include** folder in your project.
4. Link your program with the **expreval** static library inside the **lib** folder.

# 4 Examples

## 4.1 Tokenizer

```
#include <iostream>
#include <string>
#include <expreval.h>

using namespace std;

int main(int argc, const char** argv){
    string text = "5+3-2";
    ExprEval::Tokenizer tokenizer(text);

    ExprEval::Engine::Node node;
    do{
        node = tokenizer.get();
        cout << node.get() << endl;
    }while(node.type != ExprEval::Engine::NodeType::Empty);

    return 0;
}
```

## 4.2 Expression

```
#include <iostream>
#include <string>
#include <expreval.h>

using namespace std;

int main(int argc, const char** argv){
    string text = "5+3-2";
    ExprEval::Expression expression;

    try{
        cout << text << " = " << expression.evaluate(text) << endl; // 6
    } catch(const ExprEval::Engine::Exception& e){
        cerr << e.what() << endl;
    }

    return 0;
}
```

## 4.3 Function

```
#include <iostream>
#include <string>
#include <expreval.h>
```

```cpp
using namespace std;

int main(int argc, const char** argv){
    ExprEval::Function f({"x", "y"}, "x*y/100");

    try{
        cout << "f(4, 50) = " << f({4, 50}) << endl; // f(4, 50) = 2
    } catch(const ExprEval::Engine::Exception& e){
        cerr << e.what() << endl;
    }

    // This function("g") uses the previous one("f")
    ExprEval::Function g({"x[0]", "x[1]"},
                         f.get({"x[0]-x[1]","80-x[1]"}) + "+x[0]*x[1]");

    try{
        cout << "g(4, 50) = " << g({4, 50}) << endl; // g(4, 50) = 186.2
    } catch(const ExprEval::Engine::Exception& e){
        cerr << e.what() << endl;
    }

    return 0;
}
```

### 4.4 ExprEval Calculator

ExprEval Calculator is a complete project for the showcase of ExprEval library. In this project, I have created a calculator with a GUI written in Python. However, the expressions are evaluated by the **ExprEval** library in C++. If you want to check it out, it is located in the **ExprEval/calculator** directory.