Arithmetic Evaluator Report

# Features

My arithmetic evaluator supports a variety of features including:

* Conversion to and from: infix, postfix, or prefix notation
* Postfix evaluation
* Numbers with more than one digit
* Floating point numbers
* Negative numbers through unary minus: ex => -**3--4 or 2 + -(5 \* 7 ^ 2)**
* 3 parenthesis types: **{[( )]}**
* Standard operators with precedence and associatively accounted for
* Implicit multiplication => **4(2 + 2) => 4 \* (2 + 2)**
* Single argument functions: ex => **sin(1 + sqrt(4)) or sin2.77**

The following operators are implemented:

* Binary addition (+)
* Binary subtraction (-)
* Unary negation (-)
* Binary multiplication (\*)
* Binary division (/)
* Binary modulo (%)
* Binary exponent (^)

## Note about unary negation:

In order to preserve correct precedence and associatively for unary subtraction without needing to use a different symbol than '-', a negation becomes a multiplication by -1.

**Example: -3--2\*-(3\*-2) becomes ((3\*-1)-((2\*-1)\*((3\*(2\*-1))\*-1)))**

Both of these evaluate to 9.

## Note about entering postfix or prefix expressions:

Both of these types of expressions **must be separated by spaces**.

Postfix Ex: **25 1 +** and not **251+**

The parser for these expressions delimits on spaces.

The following functions are implemented:

* sin (radians)
* cos (radians)
* tan (radians)
* asin (radians)
* acos (radians)
* atan (radians)
* log (base 10 logarithm)
* ln (base e logarithm)
* floor
* ceil
* sqrt
* abs
* n! (factorial of n)
* sum (summation formula (n\*n+1) / 2)

Additional functions can be easily added. Functions are added as function pointers in the constructor:

//add in some standard functions (string, function pointer)

m\_functions.push\_back(make\_pair("sin",sin));

# Implementation

The most important components of the implementation are the following 2 methods:

vector<string> tokenizeInfix(const string& input);

string inToPost();

Tokenizing the infix string is an important component. It will parse numbers, functions, operators, and parenthesis. It will also add in implicit multiplication.

Once this is done, the inToPost method is arguably the most important one. It implements the Shunting-Yard algorithm: <http://en.wikipedia.org/wiki/Shunting-yard_algorithm>

This algorithm is similar to the one seen in class, however it allows for more advanced features such as associatively, and functions.

In order to promote code reusability and to simplify adding more features, I convert from infix to prefix by converting from infix to postfix then from postfix to prefix notation. This is because postfix to prefix is much simpler than the reverse of the shunting yard algorithm.

Conversion from prefix/postfix back to infix is a simple stack based approach that produces a fully parenthesized infix expression. Nothing is done to optimize useless parenthesis.

## Evaluation

Evaluation is simple. Postfix is evaluated using a stack based approach. If it is a binary operator, 2 numbers are popped off the stack. If it is a function, only 1 is popped, and the function pointer is called.

## Changes to the Stack

The major change to the Stack implementation was to move its implementation all to the header file. Templates in C++ compile a new class for each type as needed. Therefore, in order for generics to be portable in C++, they should not be placed in implementation files. That is, unless you explicitly state the types you want ahead of time which is not very useful for a generic Stack.

The following article explains this:

<http://stackoverflow.com/questions/495021/why-can-templates-only-be-implemented-in-the-header-file>

The other change was to add a way to know the size of the stack. This was important for error checking.

# Error Handling

The program should never crash. Instead, when an expression is constructed, it is constructed in a try block. Various methods will throw exceptions if the expression is invalid. For example, verification is made for mismatched parenthesis.

The user should then check if **expression.hasError().** In addition, if the expression is deemed valid, it still might not evaluate correctly. Evaluation should be put in a try/catch block. For example, a division by zero may have been detected.

# Running the Program

The program can run in normal mode or silent mode. Silent mode is triggered by passing command line arguments. Silent mode can be useful for batch scripts where only a single line for the result or error is desired. It can also be useful to pipe output.

The command for silent mode is:

**input\_type output\_type expression**

For example:

**infix value "2 + 2 ^ sin(2) \* sqrt(2)"**

This will take in the infix expression and return the value 4.65607.

Typing 1 or 2 arguments at the command line will give usage information.

To use it in normal mode, simply execute it and follow the on screen instructions.