Introduction

I am pleased to present a runtime "expression evaluator" (parser) which is intended to allow you to defer to run-time calculations (like, how many seconds a weapon takes to reach its target).

The intention here is that, rather than hard-coding things like:

```
power_of_weapon = str * 3 + dex / 2
```

... you could put them into "area files" (or the equivalent) and have them evaluated by the MUD server.

This is done by processing a "string" expression (ie. text) and producing a result. The calculations are done using floating-point numbers (doubles) to give maximum accuracy and allow for fractional results.

```
Example:  2 + 2
Result: 4

Example:  log10 (1234)
Result: 3.09132
```

The parser builds up results "on the fly" reporting syntax problems by throwing an exception.

```
Example: 1 + 3 + )
exception: Unexpected token: )
```

Run-time errors (principally "divide by zero") are also thrown as an exception:

```
Example: 2 / 0
exception: Divide by zero
```

Supported operations

Basic arithmetic: + - / *

```
Example: 1000 + 123
Result: 1123

Example: 44 * 55
Result: 2420
```

Expressions are evaluated from left-to-right, with divide and multiply taking precedence over add and subtract.

```
Example:  2 + 3 * 6
Result: 20   (the multiply is done first)
```

Parentheses can be used to change evaluation order.

```
Example:  (2 + 3) * 6
Result: 30  (the add is done first)
```

Whitespace is ignored. However 2-character symbols (like ==) cannot have imbedded spaces.

Symbols

The parser supports an unlimited number of named symbols (eg. "str", "dex") which can be pre-assigned values, or assigned during use.

```
Example (in C++):

  Parser p;   // create a parser instance
  p ["str"] = 55;   // assign value to "str"
  p ["dex"] = 67;   // assign value to "dex"
  double result = p.Evaluate ("str + dex * 2");   // use in expression
  double str = p ["str"];   // retrieve value of "str"

Example: a=42, b=6, a*b
Result: 252
```

There are two built-in symbols:

```
 pi = 3.1415926535897932385
 e  = 2.7182818284590452354
```

Symbols can be any length, and must consist of A-Z, a-z, or 0-9, or the underscore character. They must start with A-Z or a-z. Symbols are case-sensitive.

Assignment

Pre-loaded symbols, or ones created on-the-fly, can be assigned to, including the standard C operators of +=, -=, *= and /=.

```
Example: a=42, a/=7
Result: 6

Example: dex = 10, dex += 22
Result: 32
```

Comparisons

You can compare values for less, greater, greater-or-equal etc. using the normal C operators.

```
Example: 2 + 3 > 6 + 8
Result: 0  (false)

Example; 2 + 3 < 6 + 8
Result: 1  (true)
```

The comparison operators are: <, <=, >, >=, ==, !=

These are a lower precedence than arithmetic, in other words addition and subtraction, divide and multiply will be done before comparisons.

Logical operators

You can use AND, OR, and NOT (using these C symbols: &&, ||, !)

```
Example: a > 4 && b > 8     //  (a > 4 AND b > 8)
Example: a < 10 || b == 5   //  (a < 10 OR b == 5)
Example: !(a < 4)           // NOT (a < 4)
```

These are a lower precedence than comparisons, so the examples above will work "naturally".

Other functions

Various standard scientific functions are supported, by using:

function (argument) or function (argument1, argument2)

Single-argument functions are:

abs acos asin atan atanh ceil cos cosh exp exp floor log log10 sin sinh sqrt tan tanh

These behave as documented for the C runtime library.

```
Example: sqrt (64)
Result: 8
```

Note that functions like sin, cos and tan use radians, not degrees. To convert to radians, take degrees and multiply by pi / 180 (the value of pi is built-in).

```
Example: sin (45 * pi / 180)
Result: 0.7071
```

Three other functions which are not directly in the standard library are:

int

int (arg) <-- drops the fractional part

```
Example: int (1.2)
Result: 1

Example: int (-1.2)
Result: -1
```

rand

rand (arg) <-- returns a number in the range 0 to arg

The rand function returns an integer (whole number) result, it will never return arg itself.

eg. rand (3): might return: 0, 1 or 2 (and nothing else)

percent

percent (arg) <-- returns true (1.0) arg % of the time, and false (0.0) the rest of the time

eg. percent (40) will be true 40% of the time

Two-argument functions are:

min (arg1, arg2) <-- returns whichever is the lower
max (arg1, arg2) <-- returns whichever is the higher
mod (arg1, arg2) <-- returns the remainder of arg1 / arg2 - throws an exception if arg2 is zero
pow (arg1, arg2) <-- returns arg1 to the power arg2
roll (arg1, arg2) <-- rolls an arg2-sided dice arg1 times.

```
Example: roll (2, 4)  (in other words 2d4)
Result: 10
```

If test

Finally you can do "if" tests by using the "if" function.

if (test-value, true-value, false-value)

```
Example:  if (a > 5, 22, 33)
Result: if a > 5, returns 22
        if a <= 5, returns 33
```

User-written functions

You can easily expand the inbuilt functions by adding more to the source code using the existing ones as an example.

Distinguishing symbols from functions

In order to allow for the case where people may need to use a symbol that happens to be the name of an inbuilt function (eg. abs = 5, pow += 3), the parser distinguishes functions from symbols by looking ahead for the parenthesis following the function name. This distinction is required otherwise if someone added a user-function one day that happened to be the name of a symbol used in many places, considerably confusion would result.

```
Example: pow = pow + 3
Result: 3    (pow is a symbol)

Example: pow (4, 3)
Result: 64   (4 to the power 3)

Example: pow = pow (4, 3), pow = pow + 1
Result: 65
```

In this example we are mixing pow (4, 3) - a function - with pow as a symbol name. Confusing, perhaps, but it works consistently.

```
Example: abc (20)
Result: (exception) Function 'abc' not implemented.
```

Using in a program

To use the parser in C++ code, use it like this:

```
#include "parser.h"

Parser p ("2 + 3 * 6");
double result = p.Evaluate ();
```

The expression is not evaluated until the Evaluate function is called. If you need change the expression to be evaluated you can do it by passing a string to Evalute, eg.

```
Parser p ();
double result = p.Evaluate ("2 + 3 * 6");
```

Variables can be fed in, or retrieved, using operator[], like this:

```
Parser p ();
p ["a"] = 22;  // value for symbol "a"
p ["b"] = 33;  // value for symbol "b"
double result = p.Evaluate ("c = a + b");
double c = p ["c"];   // retrieve symbol "c"
```

This effectively lets you not only return a result (the evaluated expression) but change other symbols as side-effects.

Using in fight calculations etc. in a MUD

Basically follow the guidelines above for doing calculations.

You could "feed in" the relevant variables from the player's stats (eg. str, dex, wis).

eg.

```
Parser p ("str * 3 + dex / 3");

p ["str"] = player->str;
p ["dex"] = player->dex;

double result = p.Evalute ();
```

Example:

A fighter fixes an arrow to his bow. He takes between 2 and 8 seconds, depending on his strength:

```
time_taken =  8 - 6 * (str / max_str)
```

In this case if his normalised strength is the maximum (1) then the calculation will effectively be:

```
8 - (6 * 1) = 2 seconds
```

However if his strength is zero, the calculation will be:

```
8 - (6 * 0) = 8 seconds
```

To throw in a bit of luck you might roll a dice. For example:

```
time_taken =  11 - (6 * (str / max_str)) - roll (1, 3)
```

This is rolling a 3-sided die once, giving a result in the range 1 to 3.

Thus the worst case would be 10 seconds (11 - 0 - 1) and the best case 2 seconds (11 - 6 - 3).

Maybe 20% of the time he has really bad luck, and takes another 5 seconds:

```
time_taken =  11 - (6 * (str / max_str)) - roll (1, 3) + (5 * percent (20))
```

The "percent (20)" part will return true (that is, 1.0) 20% of the time, which will then be multiplied by 5 to add on another 5 seconds.

To use other variables you might add in dexterity, like this:

```
time_taken =  8 - (6 * (str / max_str)) - (dex / max_dex)
```

Since "dex / max_dex" will be in the range 0 to 1, then this might shave off another second for very dextrous players. For players with 50% dexterity it would shave off half a second.

Downloading

The parser is available for anyone to use, at no charge, It is written in C++ and uses STL (the Standard Template Library).

Supported compilers

It has been compiled without errors or warnings under:

- Cygwin (Windows) - gcc (GCC) 3.3.1 (cygming special)
- Microsoft Visual Studio 6.0
- Linux - gcc (GCC) 3.2.2 20030222 (Red Hat Linux 3.2.2-5)
- Mac OS/X 10.3.5 (Panther) - gcc (GCC) 3.3 20030304 (Apple Computer, Inc. build 1495)

To compile the test program under Cygwin, Linux or Mac OS/X, just run the Makefile (ie. type "make"). To compiler under Visual Studio open the project file parser.dsw.

Using

If you compile as described above you will have a test program (parser, or parser.exe under Windows) which you can run and enter expressions into to test. Note that under Visual C++ it seems to be "a line behind" due to a bug in the standard library. In other words, you need to type 2 expressions before it will evaluate the first. This is only a problem in the test program which you could fix by changing the line:

```
getline (cin, inputLine);
```

To use in your own (C++) programs, simply do this:

```
#include "parser.h

// and further on

Parser p ("2+2");    // or whatever

double result = p.Evaluate ();
```

The test program test.cpp can be used as an example of using it in another program.

I'll just emphasise that the parser presented above is a completely stand-alone piece of code.

It does not require any additional libraries or files to be installed (such as flex, bison, yacc or whatever).

The exception is that you naturally need the "standard" C++ library, including the Standard Template Library, however that is supplied as standard with the compilers mentioned above, and hopefully with all modern compilers.

To use it, you simply add two files to your project:

- **parser.cpp** - the implementation of the expression parser (622 lines). You would add this to your project (makefile).
- **parser.h** - the header file, which you would use in any source files that need the parser (133 lines).

Comma operator

Some of the examples above use the "comma operator" without really explaining it.

In C, you can separate an expression into parts by using the comma operator. eg.

```
a=10, b=20, a + b
```

This effectively breaks the expression into sub-expressions, as the comma operator has the lowest priority. This means that everything between the commas is done first.

However, as the expression is evaluated left-to-right, what the above example does is:

- Assign 10 to a
- Assign 20 to b
- Add a to b

Thus, the result of that expression is 30.