To manipulate expressions, you are given a series of classes and functions, as well as some extra (optional) stuff to make writing algorithms easier, and written code clearer.

The class “elem” (short for Element)

# The Basics

An elem represents an element of an expression. Anything that can be referred to as a whole value. For example:

“x” “f(3)” “24” “(x^2)” “-y” “log7(49)”

Are all valid expressions.

But:

“x+” “4 = 2+2” “log4”

Are all invalid.

# Assignment and Movement

elem a(/\*something\*/);  
elem b(/\*something\*/);

a = b; //copies the value of “b” into “a”  
a = std::move(b); //steals all resources of “b”, that aren’t used by nodes, into “a”.

elem c(a); //copies the value of a  
elem d(std::move(a)) // steals all resources of “b”, that aren’t used by nodes, into “a”.

a << b; //almost the same as “a = std::move(b)”  
a >> b; //almost the same as “b = std::move(a)”  
//the only difference is that this will steal all owned nodes too.

There are more ways to initialize an expression, but those require any one of the node classes, which have not been explained yet.

# Comparison

An elem can be compared to another elem with == or != :

elem(“4”) == elem(“4”) //returns true

elem(“x”) != elem(“y”) //returns true

note that this comparison ignores all mathematical properties:

elem(“3+4”) != elem(“4+3”) //returns true

elem::str() returns a string which displays its contents. By default the string looks ugly, as it doesn’t take advantage of order of operations, but the printing ruleset can be customized:  
printing\_rule\_set p;  
//edit contents of p  
some\_elem.str(p); //possibly more readable or useful string

The Class “node”

# The basics

A node itself is almost practically an elem\*, although it can’t simply point to any elem. Any node can only point to data without a scoped lifespan. Nodes can be assigned to each other similarly to shared pointers, and when no more nodes point to an elem, it is destroyed. An elem can release its data with “node elem::operator~()”:

elem e(“3x+2”);

node n = ~e; // “e” is now empty, and its old resources are now managed by “n”.

\*n = elem(“7”); //we just assigned “7” to the data which “n” points to.

All of this resource talk is a bit low level, and is really just to help make more sense of why nodes work the way which they do. Just remember:

\*n = elem(“4”) //the data pointed to by “n” is now equal to “4”

n = ~elem(“4”) //create some new data equal to “4” and have n point to it.

The first one is like assigning “4” to the memory pointed at by a pointer, and the second one is like assigning newed memory with the value “4”

# The Use of node in Expression Trees

Now, nodes are what are held by multi part trees to actually let them be trees. For example:

Elem e(“4+x”);

e is made up of an operator name (plus), and two nodes. One points to the left-hand side, and one points to the right hand side. The left-hand side holds a single value “4”, and the right-hand side holds a single value “x”

e = elem(“(2\*y)/(z-7)”)

Now, e points to an operator name (division), and two nodes. The left hand side itself also points to an operator name (multiplication) as well as a node holding “2”, and one holding “y”. The right-hand side of e points to an operator name (subtraction) and a node holding z, as well as a node holding 7. You get the point.

One other note is that a function can have arbitrarily many nodes: elem(“sum(7, 4, 6)”))

nodes are also how you initialize an elem as an operation or function:

elem(~elem(4), oper\_name::sub, ~elem(‘x’)) //an operation: (4-x)

elem(“median”, { ~elem(4), ~elem(8), ~elem(6) }) //a function: median(4,8,6)

of course, you can also use already existing nodes:

node n = ~elem(“x”);

node n2 = ~elem(2);

elem e { n2, oper\_name::log, n) //log2(x)

now if you were to edit the elem used by n or n2, it would change the contents of e too.

# Reading and Writing to Elements

Although you can directly assign to an element, and get its string, you aren’t able to edit the contents in any other way, nor can you check its type directly, without it being the data of a node. Although it may make sense to be able to, there are multiple, more practical, reasons as to why you can’t/don’t.

There are 5 subclasses of node. Each one represents a different kind of element that it might point to: constant(whole number), variable(character), operation(two nodes and operator name), function(name and list of nodes), and empty(able to contain message).

They each have operator bool(), which returns true when the elem that they point to, is of their corresponding type. They all also have self-explanatory get\_something() and set\_something() member functions which throw an invalid\_type\_access exception if a type is invalidly accessed. If something invalid is assigned, it will turn the type into an empty with an error message.

Any non-comparison member function of any node class will throw a bad\_nullnode\_access if the node is null. This can be checked for by comparison to the global constant “nullnode”.