Problem 3. Expression

Part I

How does a compiler parse your code? The answer might be a little bit complex. In this problem we will be dealing with something related and easier: arithmetic expressions involving addition, subtraction, multiplication, division, and negation. An expression can be represented as a tree structure, which is named an 'expression tree' or 'abstract syntax tree'. For example, the tree representing $-(5/6) \times (3+4)$ should be:

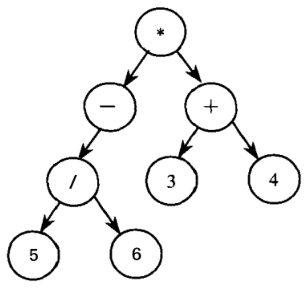

Figure 1: $-(5/6) \times (3+4)$

In an expression tree, each node is either an operator node or a number node. For each operator node, there will be edges pointing to its operands. Note that the operator '$-$' might be either a subtraction operator as in '$3-2$', or a negation operator as in '$-(2+3)$'.

We want to be able to create such a tree by calling appropriate functions, and then to evaluate the expression or to print a fully parenthesized representation of the expression. You only need to focus on the modeling of the tree and different kinds of nodes, without caring about how to parse an expression. (You will learn how to build a parser in CS131.)

We will design a **reusable** object-oriented solution to this problem. Different kinds of nodes are represented by different classes, all of which are derived from an abstract base class:

```
class Expr_node {
 public:
  Expr_node() = default;
  virtual double eval() const = 0;
  virtual std::string to_string() const = 0;
  virtual ~Expr_node() = default;
  Expr_node(const Expr_node &) = delete;
  Expr_node &operator=(const Expr_node &) = delete;
};
```

- The pure virtual member '`eval`' returns the value of the expression. The pure virtual member '`to_string`' returns a fully parenthesized representation of the expression. Of course, both of them should be allowed to be called on a constant object, so the `const` qualifier is necessary.
- This class should have a defaulted virtual destructor, since it is a polymorphic base class.

- The copy operations are deleted and you don't need to care about them in the derived classes either. This is because the meaning of copying such an object is not so clear: should it copy the entire subtree or just one node? The former is time-consuming, while the latter may cause unexpected results.

Next, we will use several classes to represent the nodes. The class '`Number_node`' represents a number node, which stores a '`double`' value. The class '`Negation_node`' represents the node of negation, and has a data member '`Expr_node *operand`' that points to the operand. The classes '`Plus_node`', '`Minus_node`', '`Multiply_node`' and '`Divide_node`' represent the binary operations, and they all have two operands pointed by '`Expr_node *lhs`' and '`Expr_node *rhs`', so we can make them derived from another class called '`Binary_node`'. The classes should be organized as in the following UML diagram:

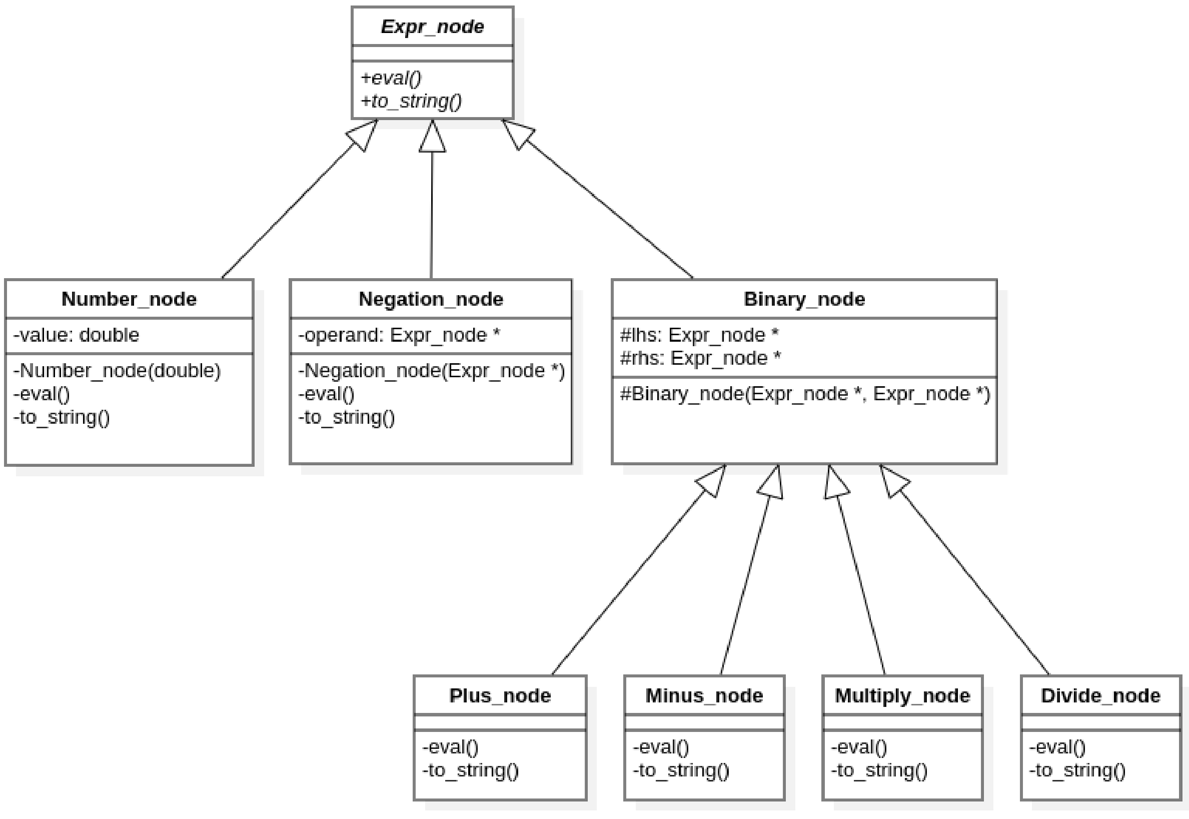

Figure 2: UML diagram for problem 3

Note that in the UML diagram, '+', '-' and '#' represent the access modifiers `public`, `private` and `protected`, respectively. All these classes are thought of as **implementation details** except `Expr_node`, so all of their members are not public. You should use the access modifiers correctly for the purpose of encapsulation. Moreover, the destructors of `Negation_node` and `Binary_node` should override the destructor of the base class, which should call `delete` on the pointers so that the entire subtree is destroyed.

Then we will design some functions as interfaces:

```
Number_node *make_number(double);
Negation_node *make_negation(Expr_node *);
```

```
Plus_node *make_plus(Expr_node *, Expr_node *);
Minus_node *make_minus(Expr_node *, Expr_node *);
Multiply_node *make_multiply(Expr_node *, Expr_node *);
Divide_node *make_divide(Expr_node *, Expr_node *);
```

These functions create an object from the given arguments, so that the user can build a tree by:

```
#include <iostream>
int main() {
  Expr_node *expr_tree = make_divide(
      make_plus(make_number(3), make_multiply(make_number(4), make_number(5))),
      make_number(6));
  std::cout << expr_tree->eval() << std::endl;
  std::cout << expr_tree->to_string() << std::endl;
  delete expr_tree;
  return 0;
}
```

The output would be

```
3.83333
((3.000000 + (4.000000 * 5.000000)) / 6.000000)
```

Since these functions need to use the constructors of the classes, we can declare them as `friend`s of the corresponding classes.

We have provided you with a template containing the definition of `Expr_node` and a prototype of `Number_node`. The rest part is for you to implement.

Notes

- Please implement everything according to the UML diagram (2) and do not modify the names of the classes and functions, or you may get compile-error.
- Your program should manage dynamic memory correctly and avoid memory leaks.
- The `to_string` function should return the fully parenthesized representation of the expression. In particular,
  - The multiplication sign is `*`, and the division sign is `/`.
  - The binary operators and its operands should be separated by a single space. But there should be no space between the unary negation operator and its operand.
  - There should always be a pair of parentheses around every sub-expression. In other words, the string returned by every `to_string` function (except `Number_node::to_string`) begins with '(' and ends with ')'. For `Number_node::to_string`, there should be a pair of parentheses if and only if the number is negative.

- To convert a `double` to `std::string`, all you need to do is to call the `std::to_string` function defined in `<string>`. This will make sure your output is identical to the answer, and you don't need to care about the precision.

Part II

Object-oriented programming has many benefits, one important of which is the **reusability**. To give a feel for this, we can try to support a new kind of expression: functions. For example, the output of

```
#include <iostream>
int main() {
  Expr_node *expr_tree =
      make_plus(make_exp(make_sin(make_number(3))), make_number(1));
  std::cout << expr_tree->eval() << std::endl;
  std::cout << expr_tree->to_string() << std::endl;
  delete expr_tree;
  return 0;
}
```

should be

```
2.15156
(exp(sin(3.000000)) + 1.000000)
```

The supported functions are `sin(x)`, `cos(x)`, `tan(x)`, `log(x)` and `exp(x)`, where $\log(x) = \ln x$ and $\exp(x) = e^x$. The corresponding interfaces are

```
/* Your_return_type */ make_sin(Expr_node *);
/* Your_return_type */ make_cos(Expr_node *);
/* Your_return_type */ make_tan(Expr_node *);
/* Your_return_type */ make_log(Expr_node *);
/* Your_return_type */ make_exp(Expr_node *);
```

where the return-types are defined on your own. You will find that you can define some other class(es) derived from `Expr_node`, without modifying anything we have written. Try to think of a solution as easy as possible!

Notes

- You should not produce more than one pair of parentheses around the argument. For example, it should be `sin(3.000000 + 5.000000)` instead of `sin((3.000000 + 5.000000))`.
- The first seven testcases are for Part I, and the last three are for Part II.

Reflection

Many interesting things can be done on the abstract syntax tree. Think about this: how can we write a program to work out the derivative of a given elementary function?

What's more, this is not the unique solution to building an abstract syntax tree. Also, there is still room for improvement. For example,

- The program is still at risk of memory leak, because the user may forget to `delete` the tree.
- The way of building an expression tree is lengthy and inconvenient. Can we implement something like this?

  ```
  Expr expr = exp((Expr(3) + Expr(4) * Expr(5)) / Expr(6));
  ```

  You can read more about this in *Ruminations on C++* Part II (Chapter 5-10), or *C++ Primer* Chapter 15 section 9.