1 Introduction

This is an exercise in object-oriented programming and polymorphism. This exercise covers the following topics:

- classes and class design
- special member functions
- move-semantics
- inheritance
- virtual functions
- `dynamic_cast`
- `typeid`
- pointers
- `new` and `delete`

This is quite a large exercise loosely based on a lab given in the course TDDC76 here at LiU. There are several purposes of this exercise; the first is to present you with a bit more challenging and hopefully fun exercise. The second reason is to demonstrate "real" object-oriented programming, and the third reason is to give you some (albeit minor) insight into how a compiler might go about parsing and handling expressions.

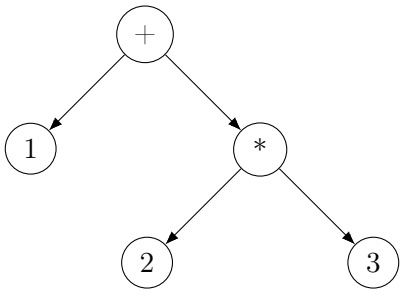
There are three parts in this exercise, each increasing in difficulty. I strongly recommend that you at least solve part a and part b.

Correct memory management is very important in this lab. Read section 3 for some tips on how to check your memory correctness using `valgrind`.

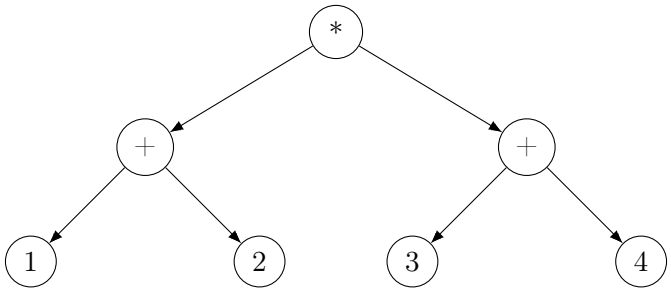
2 Calculator

In this exercise you will implement the engine of a simple calculator. The engine is represented as an *Abstract Syntax Tree* where each node either represents a number or some operator, henceforth called *Expression Tree*.

Example The expression `1+2*3` will be represented as:



Example The expression `(1+2)*(3+4)` will be represented as:



This type of tree representation doesn't need to keep track of any parenthesis since the evaluation order is specified by the tree. Notice that whenever `*` or `/` has a child which is either a `+` or a `-` there where parenthesis around that subtree in the original expression. More generally speaking, if an operator has a child node with higher precedence, it is the same as parenthesising the subtree.

There are 2 distinct categories of nodes:

Numbers Represent a single number, when evaluated this will simply return the number it represents.

Operators Correspond to one of the following operators: `+`, `-`, `*` and `/`. Must keep track of what subtrees are to the left and to the right of the operator. These subtrees should be represented as pointers to other nodes. When evaluated this node should recursively evaluate the left and right subtree, and then pass the resulting values into the operator.

There is a lot of given code for this exercise. A main program which parses the command-line arguments and calls appropriate functions is given in `Calculator.cc`. The program which turns a string containing an expression into a expression tree is given in `Parser.h` and `Parser.cc`. None of these files should require you to modify them in any way.

The file `Expression.h` contains the class `Expression` which represents a full expression tree. All of its member functions should be implemented in `Expression.cc`.

a) The Expression Tree

Things covered in this part:

- Classes
- Inheritance
- Polymorphic types
- Virtual functions
- Streams

This first part of this exercise will have you implement a class hierarchy to represent an expression tree. Each type of node should be implemented as a class with (at least) three virtual functions:

`evaluate` A function which calculates the value of the tree stored in this node. If this function is called on a number, it should return its value. If it is called on an operator, each subtree should be evaluated and then the operator should be applied to the values of the left and right subtree. This function should be overloaded for numbers and each operator.

`print_tree` What would be the point of us implementing a fancy expression tree if we can't look at it in all of its glory? This function should print the expression tree into the terminal. Do note that it should not be printed top-to-bottom, but instead left-to-right (of course, you are free to print it however you choose, but left-to-right is a lot easier than top-to-bottom).

Example The expression tree of `(1+2)*(3+4)` should be printed as:

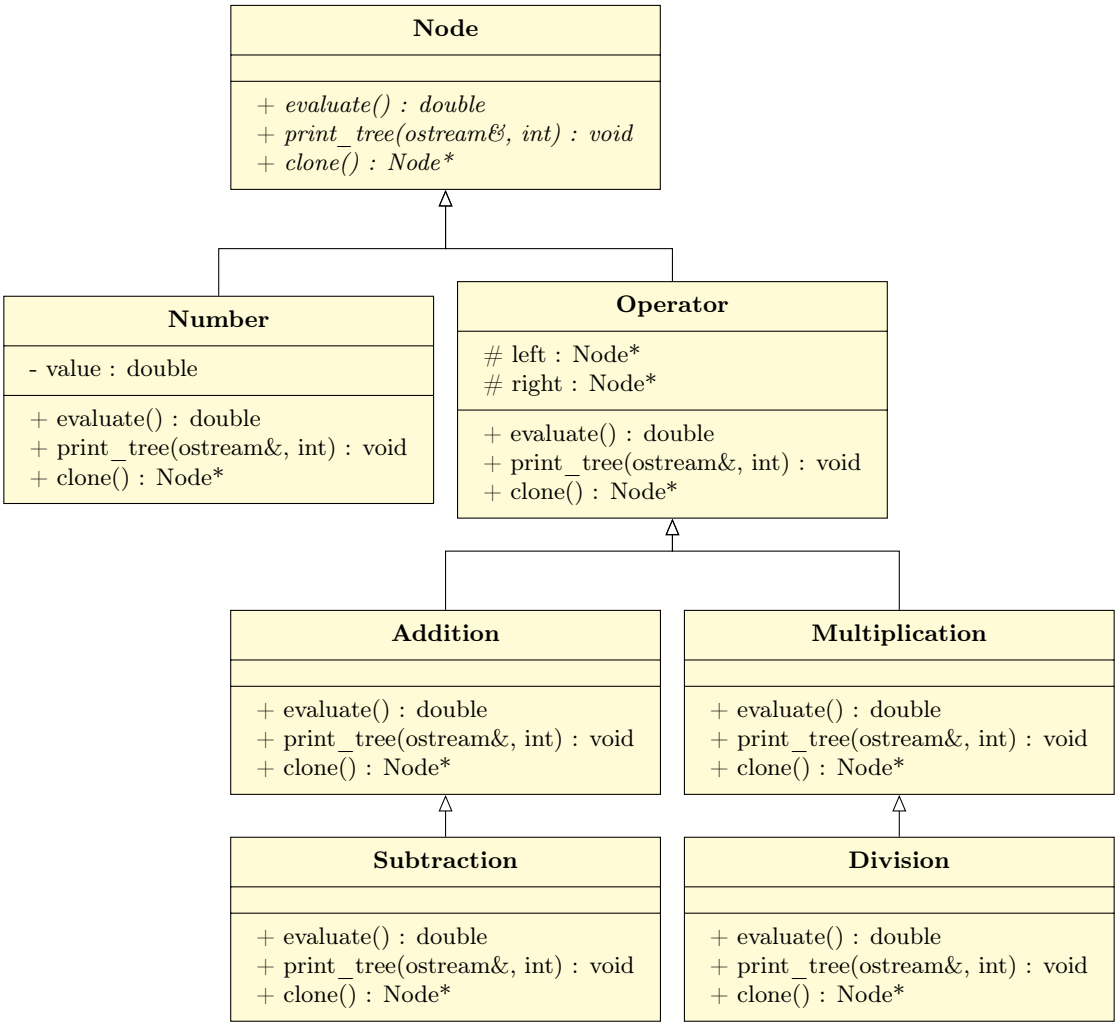
```
        1
      /
    +
      \
        2
  /
*
  \
        3
      /
    +
      \
        4
```

`print_tree` should take two parameters, a `std::ostream&` which contains a reference to the stream the tree should be printed to, and an `int` which represents how many spaces of indentation the current subtree should be printed as. The `int` should have the default value of `0`.

Hint To create indentation, use the `std::setw` function defined in `<iomanip>`.

`clone` This function should return a *deep* copy of the subtree stored in this node. Since trees are highly recursive structures, it is recommended to implement this function recursively.

To complete part (a) of this exercises you need to implement the following class hierarchy in `Node.h` and `Node.cc`:



To compile your program you must compile all available `.cc` files. If you have `Make` available (should be available if you are on Linux), then you can run `make` in the directory which contains all the files to automatically compile the entire program.

The compiled program have several command-line arguments available. In this part you will only implement the `--print-tree` feature. There is a `--help` command if you want to see what other arguments there are. If you simply want to evaluate the expression, no arguments should be passed.

The program will start with a prompt; here you can enter any mathematical expression, and the program will either calculate its value, or print a tree (depending on the given command-line arguments). Some suitable tests are the following expressions:

```
1+1     (1-2)*3     1/2*3     (1+2)/(3-4)*-1     1.5/1.5*3.1415
```

If you get tired of entering these expressions every time, you can run the following command to directly pass in an expression into the program:

```
$ echo "1+1" | ./calc
```

b) Turning trees into expressions

Things covered in this part:

- RTTI
- `dynamic_cast`

In this part you are to add functionality which converts your expression tree into an textual expression and print it to the terminal. To do this, you should add a virtual function `void print(std::ostream& os)` to the `Node` hierarchy. The expression this function print should be correctly parenthesized, and be equivalent to the expression entered by the user (the only difference being that any reduntant parenthesis have been remove).

- `Number::print` should simply print the number;
- the `print` function of each operator should print the left subtree, the operator and then the right subtree. If the current operator is multiplication or division and one of its subtrees is either addition or subtraction, that subtree should be parenthesized.

The function `Expression::print`, implemented in `Expression.cc` should be rewritten so that it uses the newly implemented `print` function to print `root`.

To check if a `Node` is a `Addition` or `Subtraction` it is enough to check if the dynamic type is compatible with `Addition`, since `Subtraction` is derived from `Addition`.

Hint: Think closely where and how you should implement the print function in such a way that it minimizes code repetition.

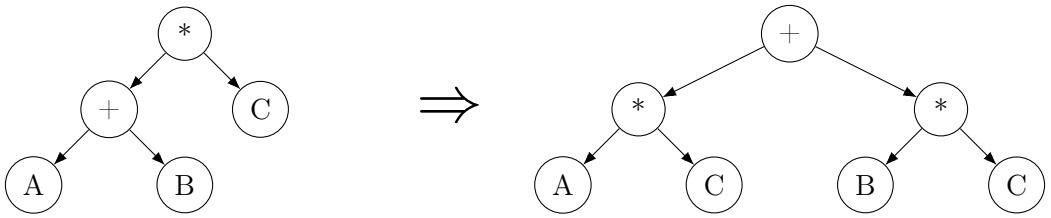
Use the `--print-expression` flag in the program to test this part of the exercise.

c) Expanding parenthesis

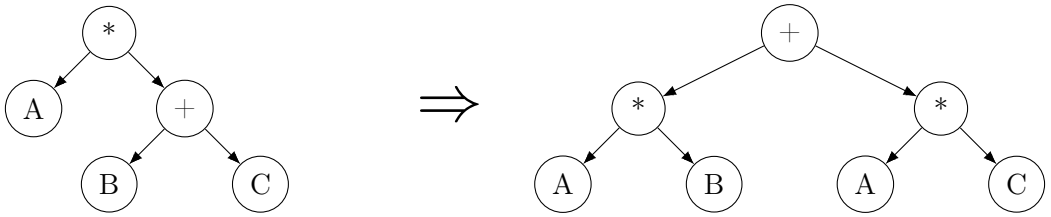
Warning: This part is very hard and does in no way reflect the difficulty level of the exam. This exercises will serve as a challenge and/or an example of a real world application of C++. You can safely skip this exercise if you find it to difficult. But I do recommend that you at least look at the reference solution.

The distributive laws in mathematics states that $(a+b)c = ac+bc$ and $a(b+c) = ab+ac$. In this exercise we will implement a feature which applies these laws to our expression tree.

We can note that these laws can be represented as operations on our tree, where we restructure the tree and add new nodes. There is one operation for each version of the law (one left and one right):



$$(A + B) * C = A * C + B * C$$



$$A * (B + C) = A * B + A * C$$

These operations does not operate in a vacuum. We do not know what A, B or C is. They might be numbers, or they might be some other complicated expressions. Therefore we should first, before we apply our transformation, recursively tranform each subtree. After that, one of the two transformations above can be applied to our current tree.

However, there is a catch. If we transform our tree and the root node of either A, B or C is an addition or subtraction, we have to transform our current node again.

The algorithm is as follows (given in C++-like pseudo-code):

```
Node* Operator::expand()
{
  lhs = lhs->expand();
  rhs = rhs->expand();
  if (this is exactly Multiplication)
  {
    if (lhs is Addition or Subtraction)
    {
      auto current = left_expand(this);
      return current->expand();
    }
    else if (rhs is Addition or Subtraction)
    {
      auto current = right_expand(this);
      return current->expand();
    }
  }
  return this;
}
```

`Number::expand()` will simply return `this`.

Your job is to translate the above pseudo-code into real, working C++. You should also implement `Operator::left_expand` and `Operator::right_expand` as `static` members of `Operator` which performs the tree transformations specified in the graphs above. These functions should return a pointer to the root node after the transformation.

You will have to rewrite `Expression::expand` such that `root = root->expand()`.

Hint: You only need to allocate one new `Multiplication` for each transformation, nothing else.

To test your program, you can run `echo "(1+2)*(3+4)" | ./calc -e -p` and check that your output is `1*3+1*4+2*3+2*4`. Of course, you will have to try expressions with nested parenthesis, but its a good start.

3 Memory Management

In this exercise it is important to manage your dynamic memory correctly. You should avoid any and all memory leaks. On Linux there is a tool called `valgrind` which can check wheter or not you have any memory leaks. To run it on your program:

```
$ valgrind --leak-check=full ./calc
```

This will run your program, but at an extremely slow rate. Enter some complicated expression, and you will after a while see something like this:

```
LEAK SUMMARY:
   definitely lost: 0 bytes in 0 blocks
   indirectly lost: 0 bytes in 0 blocks
     possibly lost: 0 bytes in 0 blocks
   still reachable: 72,704 bytes in 1 blocks
```

If you see that `definitely lost`, `indirectly lost` and `possibly lost` all are `0`, then you have no memory leaks. Everything else would probably be a memory leak, so you should check that all your allocated memory (`new`) is deallocated (`delete`).

The easiest way to guarantee correct memory usage is to rewrite the entire program such that it uses `std::unique_ptr` instead. If you find that you have a hard time fixing your memory leaks, this might be a viable option.