ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

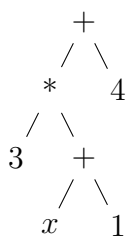
Programming Techniques for Scientific Simulations Exercise 12

HS 12
Prof. M. Troyer

Problem 12.1 Template Meta-Programming

The purpose of this exercise is to understand and complete the attached code for expression templates and algorithmic derivation.
Expression templates were explained last week in the lecture. The key insight is that the tree associated with any expression can be represented with types at compile-time; the compiler can then perform powerful optimizations on the expression or, under specific circumstances, evaluate the whole expression or parts of it at compile-time.
Heres an example of such a tree:

$$3 * (x + 1) + 4$$



Note that it consists of vertices representing some operation and leafs representing either a constant or the independent variable.
Algorithmic derivation relies on the observation that for a given tree representing an expression, one can easily calculate the derivative of the expression using the chain rule and the product rule.
We want to combine this into a very simple code that is able to calculate expressions and derivatives thereof, where the only **allowed operations are addition and multiplication** and we allow only **one independent variable**, which may however occur several times. The basic ingredient is the class `Expression` that represents vertices of the tree, which has three template parameters:

- The type of the left-hand vertex,
- the type of the right-hand vertex,
- the operation (from an `enum`).

Additionally, we have two classes for the leafs, which can be either a constant or a placeholder for the independent variable.
Each class must implement two functions:

- `T operator()(T y)`, which calculates the value of the part of the expression that follows below that vertex at $x = y$,
- `T derivative(T y)`, which calculates the value of the derivative of the part of the expression that follows below that vertex at $x = y$, obeying chain and product rule.

All this is being represented by types, i.e. the type of the above expression would be:

```
Expression<
        Expression<Constant<int>,
                Expression<Placeholder, Constant<int>, Plus>,
                Mul >,
        Constant<int>, Plus >
```

Try to understand the published code (see the website or repository) and fill in the missing lines in the `derivative` functions of the `Constant`, `Placeholder` and of the `Expression` such that it obeys chain and product rule.
Two source files are given on the website: `main.cpp` for compiling into a full program, and `algorithmic_derivation.h` containing the meta-programming part. For illustration purposes, the assembler output for the example is also in the archive. You can see that with a smart compiler and the right compiler options (`-O3`), everything is evaluated at compile-time; the assembly[1] therefore contains the final numbers ($f(8) = 974$ and $f'(8) = 157$).

[1] Use compiler flag `-save-temps` to store the assembler output permanently.