Middle East Technical University

Department of Computer Engineering

CENG 242

Programming Language Concepts
Spring 2023
Programming Exam 4

Due: May 21th 2023 23:55
Submission: **via ODTUClass**

1 Story

Alice's Adventures in Computerland...

As Alice roamed through Computerland[1], she chanced upon a sagacious caterpillar perched atop a toadstool. The caterpillar was renowned throughout the land for his deep wisdom in mathematics and computer science.

Curious, Alice inquired as to the nature of the caterpillar's ruminations. The caterpillar replied that he was contemplating a method to approach polynomials by infinitesimal degrees, with the aim of efficiently computing their derivatives. Alice was fascinated and begged the caterpillar to elucidate.

The caterpillar explained that this method was known as automatic differentiation, whereby derivatives of mathematical functions are approximated without direct calculation. He opined that Alice could implement this method through the principles of object-oriented programming, by representing polynomials as objects and employing inheritance and polymorphism to construct a supple and extensible solution.

Alice was intrigued by the caterpillar's exposition, but also disconcerted by the prospect of implementing such a solution. The caterpillar reassured her, exhorting that with perseverance and determination, she could accomplish anything she desired.

With renewed vigour, Alice embarked on the creation of an object-oriented solution for approaching polynomials by infinitesimal degrees. She spent long hours studying mathematical concepts and programming techniques, and sought the counsel of the caterpillar for guidance.

[1] a land inherited privately from the Wonderland

Finally, after much exertion and assiduity, Alice succeeded in developing a solution that approached polynomials by infinitesimal degrees and efficiently computed their derivatives through automatic differentiation. The solution was elegant and straightforward, yet powerful and extensible. Alice was overjoyed with the outcome, and knew that the caterpillar's erudition had empowered her to achieve something truly remarkable.

From that day forth, Alice was esteemed throughout Computerland for her mastery in mathematics and computer science. She persisted in her quest for knowledge and innovation, using her skills to craft ingenious solutions for the challenges she encountered. And whenever she required guidance or inspiration, she knew that she could always turn to her sage friend, the caterpillar, for succour.

2 Introduction

2.1 Problem Definition

In this programming exam, you will be using object-oriented paradigm (polymorphism, encapsulation, abstraction, and inheritance) to construct an automatic-differentiation system (auto-grad) which takes expression trees of polynomials (parsed implicitly thanks to operator overloading) and produces another expression tree representing the result of few complex operations like the gradient with respect to a variable or the evaluation under some constrains. To this end, you will be implementing given C++ classes inherited from an abstract class and overload it's four pure methods.

2.2 Simple Example

For example, as you finish this homework, the expression in the figure 1, which **(2 * y) + x** consists of two binary operators (addition and multiplication) and two variables (x and y), should be parsed **implicitly** during compilation (i.e. without writing an extra parsing code) due to (mainly) overloaded operator+ and operator* methods in the abstract class.

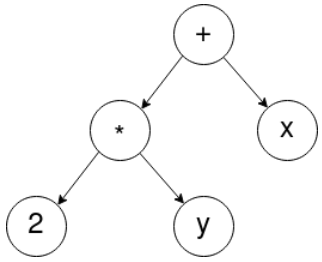

Figure 1: Parse tree

```
Sym::Var x = "x", y = "y";
Sym::Expr f = 2.0 * y + x;
```

This third-party syntactic-sugar will be explained in the following section 3. But roughly speaking, overloading **operator+** and **operator*** in the leaf class **Sym::Expr** let's the compiler to apply the following replacement

```
Sym::Expr f = Sym::Expr::operator+(Sym::Expr::operator*(2, y),x);
```

You will see in the implementation details that **operator*** and **operator+** are defined as a **friend** method of the class **Sym::Expr**. In the parameter passing, the first argument is replaced by a **Sym::Const(2)** and set to the left hand side of the binary operator **Sym::MulOp**. Additionally, **Sym::Var x("x")** is set to the right hand side. Lastly, The addition operator **Sym::MulOp::operator+** is called by the returning instance. The method

sets the instance to the left hand side of a **Sym::AddOp** instance. Finally, **Sym::Var y("y")** is set to the right hand side and the addition instance is returned. The result is set to a **Sym::Expr** instance **f**.

```
auto f_y = f.diff(y);
auto sanity =  (f_y == 2 ? "pass" : "fail");
std::cout << "check: " << sanity << std::endl;
std::cout << f_y.eval() << std::endl;
```

```
check: pass
2
```

As you can see above, we first take the gradient of the expression stored in **f** with respect to **y**, then apply a sanity check to make sure that our result is correct, then we stream it to the standard output. Note here that we also overloaded the way the expression trees are streamed so that we can print those trees like nifty mathematical lines.

2.3 General Specifications

- You simply implement all method definitions in source files given to the corresponding declarations (prototypes) in the header files. Make sure that your implementations comply with those prototypes and that you have never modified those declarations at all.
- Note that during your homework's evaluation, the header files **will be replaced with** what were given to you. Therefore, you may define helper function(s) **only in the source files** as needed.
- Codes are intended to be memory leakage-free. That means, the programmer is responsible for releasing every unit of memory that was previously allocated. However for this homework, we will first check if your code gives the intended result. As long as your code gives the correct result but cause also memory leaks, we will accept it. Otherwise you will be given 25% bonus.

$$\text{bonus} = 0.25 \times \text{grade} \times \frac{\#\text{leaked cases}}{\#\text{total cases}}$$

3 Implementation

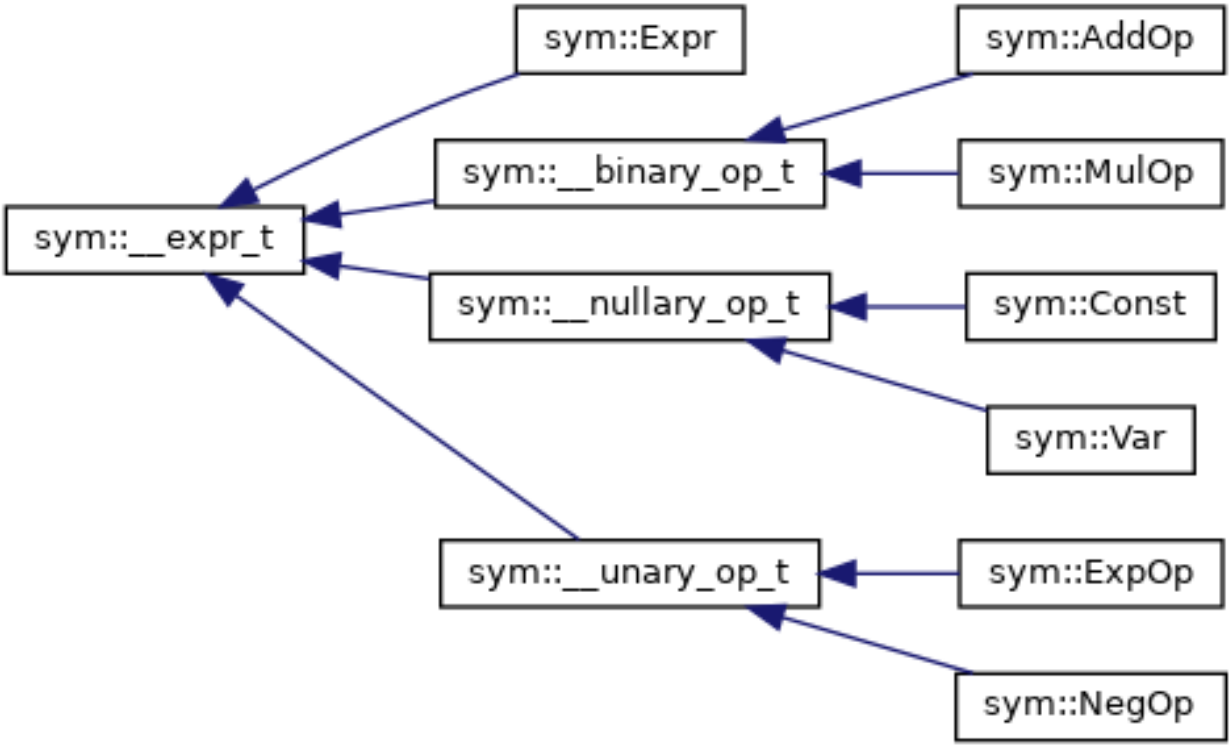

Figure 2: Class Hierarchy Diagram

This homework includes, private and protected attributes, pure and virtual methods, overloaded methods, abstract classes, and inherited classes extensively. In brief, the figure 2 shows class hierarchy by inheritance. As a naming convention, the double underscore prefix indicates an abstract class like an interface in Java.

The root abstract class **sym::__expr_t** has the following four pure methods evaluation, differentiation, visualization, and comparison as in the figure 3. Their intended behavior is attached in a comment. We expect you to implement them in the leaf classes.

There are also three abstract children classes derived from the root class, which are nullary operator (3.1), unary operator (3.2) and binary operator (3.3). These classes have some extra polymorhic methods like **is_nullary** to check of type of an instance in run-time.

Eventually this tree in figure 2 has seven leaf classes called Constant (3.1.1), Variable (3.1.2), Negation (3.2.1), Exponentiation (3.2.2), Addition (3.3.1), Multiplication (3.3.2), and Expression. Roughly speaking, for example, Addition (**sym::AddOp**) overrides such four pure functions; to act like addition, i.e. the additive group of real numbers.

```
// Evaluate the expression using the provided variable map.
// Returns a pointer to the resulting expression tree.
virtual __expr_t* eval(const var_map_t& var_map) const = 0;

// Differentiate the expression with respect to the provided variable.
// Returns a pointer to the resulting expression tree.
virtual __expr_t* diff(const std::string& v) const = 0;

// Write the expression to the output stream.
// Returns a reference to the output stream.
virtual std::ostream& operator<< (std::ostream &out) const = 0;

// Compare two expressions for equality.
// Returns true if the expressions are equal, false otherwise.
virtual bool operator==(const __expr_t& other) const = 0;
```

Figure 3: Pure Functions of root abstract class

As a proof of concept, Addition and multiplication is enough to construct an algebraic ring on real numbers so that we may work with the polynomials with finite natural powers. Additionally, exponentiation prevents us from infinite recursions thanks to Taylor's expansion $e^x = \sum \frac{x^n}{n!}$.

As a recall, we added differentiation rules for all unary and binary operators as much as needed. In these rules, we preferred the Hamiltonian notation ∇f (nabla or del operator) to denote derivative of any expression f. It is just an alternative to show it in Leibnizian notation $\frac{df}{dx}$, Euler notation $D(f)$, Newtonian notation $\dot{f}$, or Lagrangian notation f' of differentiation. Even though nabla del operator usualy denotes gradient or transpose jacobian operator as a vector of expressions, we used it for our scalar expressions. Please don't confuse.

Lastly, **for every derived class** from the root in the figure 2, **you need to implement** a run-time checker to check if an instance, pointed to by a root class's pointer (i.e. **sym::__expr_t ***), is of that derived class or not, during execution. For details please see advice 6 in the section 5 tips and tricks.

3.1 sym::__nullary_op_t

The **sym::__nullary_op_t** is an abstract class directly derived from **sym::__expr_t** that represents nullary operators, i.e. operators that take no arguments. It's a baseline to inherit **sym::Const** (3.1.1) and **sym::Var** (3.1.2). Note also for those derived classes, you need to implement a getter as well as an explicit casting.

3.1.1 sym::Const

(5 Points): Here in this class, you will implement the following methods,

- **sym::Const::Const**: (constructor) takes a **double** as an argument and sets it to its attribute (let's say) **variable**.

- **sym::Const::get_value**: (getter) returns the **variable**.
- **sym::Const::operator double**: (cast operator) returns the **variable**.
- **sym::Const::eval**: (overridden method) returns a pointer to a new **sym::Const** instance with the same **variable**.
- **sym::Const::diff**: (overridden method) returns a pointer to a new expression tree which is the derivative of **this** expression tree with respect to its input **v**.
- **sym::Const::operator<<**: (overridden method) writes **variable** to its input "output stream" and returns it.
- **sym::Const::operator==**: (overridden method) compares if its input (say **other**) is a **sym::Const** instance with the same variable as it's **variable**.

The corresponding header file is in this form:

```
// A symbolic constant
class Const : public __nullary_op_t {
public:
    Const(double value) : value_(value) {}

    // getter and setter methods for class attributes
    double get_value() const;
    operator double() const;

    // overriden methods for instance check via run-time polymorphism
    bool is_con() const override;

    // Evaluate the expression using the provided variable map.
    // Returns a pointer to the resulting expression tree.
    __expr_t* eval(const var_map_t& vars = var_map_t()) const override;

    // Differentiate the expression with respect to the provided variable.
    // Returns a pointer to the resulting expression tree.
    __expr_t* diff(const std::string& v) const override;

    // Write the expression to the output stream.
    // Returns a reference to the output stream.
    std::ostream& operator<< (std::ostream &out) const override;

    // Compare two expressions for equality.
    // Returns true if the expressions are equal, false otherwise.
    bool operator==(const __expr_t& other) const override;

private:
    double value_;
};
```

3.1.2 sym::Var

(5 points) Here in this class, you will implement the following methods,

- **sym::Var::Var**: (constructor) takes a **std::string** as an argument and sets it to its attribute (let's say) **variable**.
- **sym::Var::get_variable**: (getter) returns the **variable**.
- **sym::Var::operator std::string**: (cast operator) returns the pointer to the **variable**.
- **sym::Var::eval**: (overridden method) returns a pointer to a new **sym::Var** with the same **variable** unless it's **variable** is in the input "variable map", otherwise it returns a constant with a **value** which is mapped by the **variable** in the input "variable map".
- **sym::Var::diff**: (overridden method) returns a pointer to a new expression tree which is the derivative of **this** expression tree with respect to its input **v**, i.e. it's a one-constant[2] if **variable** equals **v**, zero-constant[3] otherwise.
- **sym::Var::operator<<**: (overridden method) writes **variable** to its input "output stream" and returns it.
- **sym::Var::operator==**: (overridden method) compares if its input (say **other**) is a **sym::Var** instance with the same variable as it's **variable**.

It's header file is in this form:

```
// A symbolic variable
class Var : public __nullary_op_t {
public:
    Var(const char* name) : var_(name) {}
    Var(const std::string& name) : var_(name) {}

    // getter and setter methods for class attributes
    std::string get_variable() const;
    operator std::string() const;

    // overriden methods for instance check via run-time polymorphism
    bool is_var() const override;

    // Evaluate the expression using the provided variable map.
    // Returns a pointer to the resulting expression tree.
    __expr_t* eval(const var_map_t& vars = var_map_t()) const override;

    // Differentiate the expression with respect to the provided variable.
    // Returns a pointer to the resulting expression tree.
    __expr_t* diff(const std::string& v) const override;

    // Write the expression to the output stream.
    // Returns a reference to the output stream.
```

[2] a constant with variable one, i.e. sym::Const(1)
[3] a constant with variable zero, i.e. sym::Const(0)

```
    std::ostream& operator<< (std::ostream &out) const override;

    // Compare two expressions for equality.
    // Returns true if the expressions are equal, false otherwise.
    bool operator==(const __expr_t& other) const override;

private:
    const std::string var_;
};
```

3.2 sym::__unary_op_t

The **sym::__unary_op_t** is an abstract class directly derived from **sym::__expr_t** that represents unary operators, i.e. operators that take one argument. It's a baseline to inherit **sym::NegOp** (3.2.1) and **sym::ExpOp** (3.2.2).

3.2.1 sym::NegOp

(15 points) Here in this class, you will implement the following methods,

- **sym::NegOp::NegOp**: (constructor) takes a expression sub-treepointed to by a root class pointer (i.e. **sym::__expr_t**) as an argument and sets it to its attribute (let's say) **operand**.
- **sym::NegOp::eval**: (overridden method) returns a pointer to a new **sym::Const** with a negative "variable" if its' **operand** is a constant with "variable". Otherwise, it returns a pointer to a new negation instance whose operand points to a sub-tree that is equal to one pointed to by **operand**.
- **sym::NegOp::diff**: (overridden method) returns a pointer to a new expression tree that is equal to the differentiation of the tree pointed to by **operand**.

$$\nabla(\text{NegOp}(op)) = \text{NegOp}(\nabla(op))$$

- **sym::NegOp::operator<<**: (overridden method) writes "-**operand**" to its input "output stream" and returns it in top-to-down fashion. Additionally, if it's **operand** is not nullary, encloses it with parenthesis i.e. "-(**operand**)", please check the advice 2.
- **sym::NegOp::operator==**: (overridden method) compares if its input argument (say **other**) is a **sym::NegOp** instance with a sub-tree that is (recursively) equal to the one pointed to by **operand**.

Here is it's header:

```
// A symbolic unary negation operator
class NegOp : public __unary_op_t {
public:
    NegOp(const __expr_t* op) : __unary_op_t(op) {}
```

```
    // overriden methods for instance check via run-time polymorphism
    bool is_neg() const override;


    // Evaluate the expression using the provided variable map.
    // Returns a pointer to the resulting expression tree.
    __expr_t* eval(const var_map_t& vars = var_map_t()) const override;

    // Differentiate the expression with respect to the provided variable.
    // Returns a pointer to the resulting expression tree.
    __expr_t* diff(const std::string& v) const override;

    // Write the expression to the output stream.
    // Returns a reference to the output stream.
    std::ostream& operator<< (std::ostream &out) const override;

    // Compare two expressions for equality.
    // Returns true if the expressions are equal, false otherwise.
    bool operator==(const __expr_t& other) const override;
};
```

3.2.2 sym::ExpOp

(15 points) Here in this class, you will implement the following methods,

- **sym::ExpOp::ExpOp**: (constructor) takes a expression sub-tree pointed to by a root class pointer (i.e. **sym::__expr_t**) as an argument and sets it to its attribute (let's say) **operand**.
- **sym::ExpOp::eval**: (overridden method) returns a pointer to a new **sym::Const** with an exponent "variable" if its' **operand** is a constant with "variable". Otherwise, it returns a pointer to a new exponentiation instance whose operand points to a sub-tree that is equal to one pointed to by **operand**.
- **sym::ExpOp::diff**: (overridden method) returns a pointer to a new expression tree that is equal to the differentiation of the tree pointed to by **operand**.

$$\nabla(\text{ExpOp}(op)) = \text{MulOp}(\nabla(op), \text{ExpOp}(op))$$

- **sym::ExpOp::operator<<**: (overridden method) writes "e^**operand**" to its input "output stream" and returns it in top-to-down fashion. Additionally, if it's **operand** is not nullary, encloses it with parenthesis i.e. "e^(**operand**)", please check the advice 2.
- **sym::ExpOp::operator==**: (overridden method) compares if its input argument (say **other**) is a **sym::ExpOp** instance with a sub-tree that is (recursively) equal to the one pointed to by **operand**.

Here is it's header:

```
// A symbolic unary exponentiation operator
class ExpOp : public __unary_op_t {
public:
    ExpOp(const __expr_t* op) : __unary_op_t(op) {}

    // overriden methods for instance check via run-time polymorphism
    bool is_exp() const override;


    // Evaluate the expression using the provided variable map.
    // Returns a pointer to the resulting expression tree.
    __expr_t* eval(const var_map_t& vars = var_map_t()) const override;

    // Differentiate the expression with respect to the provided variable.
    // Returns a pointer to the resulting expression tree.
    __expr_t* diff(const std::string& v) const override;

    // Write the expression to the output stream.
    // Returns a reference to the output stream.
    std::ostream& operator<< (std::ostream &out) const override;

    // Compare two expressions for equality.
    // Returns true if the expressions are equal, false otherwise.
    bool operator==(const __expr_t& other) const override;
};
```

3.3 sym::__binary_op_t

The **sym::__binary_op_t** is an abstract class directly derived from **sym::__expr_t** that represents binary operators, i.e. operators that take two arguments. It's a baseline to inherit **sym::AddOp** (3.3.1) and **sym::MulOp** (3.3.2).

3.3.1 sym::AddOp

(25 points) Here in this class, you will implement the following methods,

- **sym::AddOp::AddOp**: (constructor) takes two expression sub-trees pointed to by a root class pointers (i.e. **sym::__expr_t**) as arguments and sets them to its attributes (let's say) **right hand side** and **left hand side**.

- **sym::AddOp::eval**: (overridden method) returns a pointer to a new expression tree (recursively) equal to the one hand side if the other side is a zero-constant. However, it returns a pointer to a constant with a variable summation of "value1" and "value2" if both attributes are pointing to a value1-constant and a value2-constant. Otherwise, it returns a pointer to a new addition instance whose attributes are pointing to two sub-trees which are equal to ones pointed by **left hand side** and **right hand side** respectively.

- **sym::AddOp::diff**: (overridden method) returns a pointer to a new expression tree that is equal to the differentiation of the tree pointed to by **left hand side** + **right**

hand side.

$$\nabla(\text{AddOp}(lhs, rhs)) = \text{AddOp}(\nabla(lhs), \nabla(rhs))$$

- **sym::AddOp::operator<<**: (overridden method) writes "**left hand side** + **right hand side**" to its input "output stream" and returns it in top-to-down fashion. Additionally, if any side is not nullary, encloses it with parenthesis e.g. "(**left hand side**) + **right hand side**" if only left side is non nullary, please check the advice 2.

- **sym::AddOp::operator==**: (overridden method) compares if its input argument (say **other**) is a **sym::AddOp** instance with two sub-trees that are (recursively) equal to ones pointed to by either sides reciprocally. For details please check the advice 3.

Here is it's header:

```
// A symbolic binary addition operator
class AddOp : public __binary_op_t {
public:
    AddOp(const __expr_t* lhs, const __expr_t* rhs) : __binary_op_t(lhs,
        rhs) {}

    // overriden methods for instance check via run-time polymorphism
    bool is_add() const override;

    // Evaluate the expression using the provided variable map.
    // Returns a pointer to the resulting expression tree.
    __expr_t* eval(const var_map_t& vars = var_map_t()) const override;

    // Differentiate the expression with respect to the provided variable.
    // Returns a pointer to the resulting expression tree.
    __expr_t* diff(const std::string& v) const override;

    // Write the expression to the output stream.
    // Returns a reference to the output stream.
    std::ostream& operator<< (std::ostream &out) const override;

    // Compare two expressions for equality.
    // Returns true if the expressions are equal, false otherwise.
    bool operator==(const __expr_t& other) const override;
};
```

3.3.2 sym::MulOp

(25 points) Here in this class, you will implement the following methods,

- **sym::MulOp::MulOp**: (constructor) takes two expression sub-trees pointed to by a root class pointers (i.e. **sym::__expr_t**) as arguments and sets them to its attributes (let's say) **right hand side** and **left hand side**.

- **sym::MulOp::eval**: (overridden method) returns a pointer to a new expression tree (recursively) equal to the one hand side if the other side is a one-constant. However, it returns a pointer to a constant with a variable product of "value1" and "value2" if

both attributes are pointing to a value1-constant and a value2-constant. Otherwise, it returns a pointer to a new multiplication instance whose attributes are pointing to two sub-trees which are equal to ones pointed by **left hand side** and **right hand side** respectively.

- **sym::MulOp::diff**: (overridden method) returns a pointer to a new expression tree that is equal to the differentiation of the tree pointed to by **left hand side** times **right hand side**.

$$\nabla(\text{MulOp}(lhs, rhs)) = \text{AddOp}(\text{MulOp}(\nabla(lhs), rhs), \text{MulOp}(lhs, \nabla(rhs)))$$

- **sym::MulOp::operator<<**: (overridden method) writes "**left hand side** * **right hand side**" to its input "output stream" and returns it in top-to-down fashion. Additionally, if any side is not nullary, encloses it with parenthesis e.g. "**left hand side** * (**right hand side**)" if only right side is non nullary, please check the advice 2.

- **sym::MulOp::operator==**: (overridden method) compares if its input argument (say **other**) is a **sym::MulOp** instance with two sub-trees that are (recursively) equal to ones pointed to by either sides reciprocally. For details please check the advice 3.

here is it's header

```
// A symbolic binary multiplication operator
class MulOp : public __binary_op_t {
public:
    MulOp(const __expr_t* lhs, const __expr_t* rhs) : __binary_op_t(lhs,
       rhs) {}

    // overriden methods for instance check via run-time polymorphism
    bool is_mul() const override;

    // Evaluate the expression using the provided variable map.
    // Returns a pointer to the resulting expression tree.
    __expr_t* eval(const var_map_t& vars = var_map_t()) const override;

    // Differentiate the expression with respect to the provided variable.
    // Returns a pointer to the resulting expression tree.
    __expr_t* diff(const std::string& v) const override;

    // Write the expression to the output stream.
    // Returns a reference to the output stream.
    std::ostream& operator<< (std::ostream &out) const override;

    // Compare two expressions for equality.
    // Returns true if the expressions are equal, false otherwise.
    bool operator==(const __expr_t& other) const override;
};
```

3.4 sym::Expr

(7 points): The **sym::Expr** is an actual class directly derived from **sym::__expr_t** that represents expressions, i.e. a wrapper to contain pointers to exprssion tree instances. Implement following methods,

- **sym::Expr::Expr**: (constructor) takes an expression tree pointed to by a root class pointers (i.e. **sym::__expr_t**) as an argument and sets them to its attribute (let's say) **root**.
- **sym::MulOp::eval**: (overridden method) returns a pointer to a new expression tree (recursively) equal to one pointed to by **root**.
- **sym::MulOp::diff**: (overridden method) returns a pointer to a new expression tree (recursively) equal to the derivative of one pointed to by **root**.
- **sym::MulOp::operator<<**: (overridden method) writes "**root**" to the input "output stream" and returns it.
- **sym::MulOp::operator==**: (overridden method) compares if its input argument (say **other**) is equal to the expression-tree pointed to by **root**.

Here is it's header

```
class Expr : public __expr_t {
public:
    Expr(const Const& c);
    Expr(const Var& v);
    Expr(const __expr_t* expr_);
    Expr(const __expr_t& expr_);
    Expr(const Expr& other);

    ~Expr();

    // derived methods for instance check via run-time polymorphism

    // Evaluate the expression using the provided variable map.
    // Returns a pointer to the resulting expression tree.
    __expr_t* eval(const var_map_t& vars = var_map_t()) const override;

    // Differentiate the expression with respect to the provided variable.
    // Returns a pointer to the resulting expression tree.
    __expr_t* diff(const std::string& v) const override;

    // Write the expression to the output stream.
    // Returns a reference to the output stream.
    std::ostream& operator<< (std::ostream &out) const override;

    // Compare two expressions for equality.
    // Returns true if the expressions are equal, false otherwise.
    bool operator==(const __expr_t& other) const override;
    bool operator==(const Expr& other) const;
```

```
private:
    const __expr_t *expr_;
};
```

3.5 overload.cpp

(8 points): This is a separate file containing overloading of the following operators and methods

- operator-(const sym::__expr_t &op)
- operator+(const sym::__expr_t &lhs, const sym::__expr_t &rhs)
- operator+(const double lhs, const sym::__expr_t &rhs)
- operator+(const sym::__expr_t &lhs, const double rhs)
- operator*(const sym::__expr_t &lhs, const sym::__expr_t &rhs)
- operator*(const double lhs, const sym::__expr_t &rhs)
- operator*(const sym::__expr_t &lhs, const double rhs)
- exp(const sym::__expr_t &op)

Here is part of **sym::__expr_t**'s header for overloading part

```
    class __expr_t {
        // ...

        friend __expr_t& operator-(const __expr_t &op);

        friend __expr_t& operator+(const __expr_t &lhs, const __expr_t
            &rhs);
        friend __expr_t& operator+(const double lhs, const __expr_t &rhs);
        friend __expr_t& operator+(const __expr_t &lhs, const double rhs);

        friend __expr_t& operator*(const __expr_t &lhs, const __expr_t
            &rhs);
        friend __expr_t& operator*(const double lhs, const __expr_t &rhs);
        friend __expr_t& operator*(const __expr_t &lhs, const double rhs);
    };

    __expr_t& exp(const __expr_t &op);
```

4 Regulations

1. **Implementation and Submission:** The template header files are available in the Virtual Programming Lab (VPL) activity called "PE4" on OdtuClass. At this point, you have two options:

- You can download the template files, complete the implementation and test it with the given sample main.cpp file (and also your own test cases) on your local machine. Then submit the same file through this activity.
- You can directly use the editor of VPL environment by using the auto-evaluation feature of this activity interactively. Saving the code is equivalent to submit a file.

If you work on your own machine, make sure that your implementation can be compiled and tested in the VPL environment after you submit it.
There is no limitation on online trials or submitted files through OdtuClass. The last one you submitted will be graded.

2. **Cheating: We have zero tolerance policy for cheating.** People involved in cheating (e.g. code similarity with others or AI models) will be punished according to the university regulations.

3. **Fairness:** Your program will be evaluated automatically using "black-box" technique so make sure to obey the specifications. No erroneous input will be used. Therefore, you don't have to consider the invalid expressions.

4. **Safety:** Your program will be analyzed by a popular memory management tool called valgrind. Therefore, it's your responsibility to **delete** or **free** memory units allocated by heap via **malloc** or **new**. If your code works without any memory leakage[4] you will be given 25% bonus grade.

5. **Grading**: You can assume for every class you have to implement that

   - The constructor(s) and destructor(s) are 20%,
   - The evaluation method is 20%,
   - The differentiation method is 20%,
   - The visualization method is 20%, and
   - The comparison method is 20%.

   of the points assigned to that class or partition.

6. **Evaluation:** It's not necessarily the case that your actual grade will be the same as the number of sample test cases passed in either VPL or local environments. We intend it to be an unbiased estimator of your actual grade however it is your responsibility to consider all cases on your work. Your implementations will be evaluated by the official test cases to determine your *actual* grade after the deadline.

7. **Communication**: There is a discussion forum sharing the same name with this homework on the odtuclass course page. No code sharing is possible on the forum so if you have a question that you cannot share to public, please email me directly.

[4] a runtime error occuring when memory's allocated from the heap but is not properly freed when it's no longer needed. such errors are detected when the last pointer or reference to the memory is reset without deallocating the memory first.

5 Some Tips and Tricks

1. As a good programming practice, please prefer to use either **malloc** and **free** or **new** and **delete**. Don't mix them up both. Otherwise this will create non-deterministic run-time errors that will make you pull your hair as approaching to the deadline :).

2. For printing any operator (including nullary, unary, and binary), please check for every operand that if it's an instance of **sym::__nullary_op_t** class, then exclude parenthesis, otherwise include parenthesis. That's it :).

3. For comparing two binary operators please consider that it's not neccesary for reciprocal operands to be equal. That means, given that two binary operators (let's say former and latter) are two instances of the same class, that the former's first operand and the latter's second operand are equal, and vice versa (that the former's second operand and the latter's first operand) are equal. Then we consider two operators to be equal.

4. The use of smart pointers for this homework is not necessary since our aim isn't losing you in details but is acceptable and may make your debuging less painful.

5. The use of dynamic casting is inevitable for especially run-time type checking and casting of polymorphic types, such as when casting from a base class pointer to a derived class pointer. It ensures that the cast is valid by checking the actual type of the object at run-time, and returns a null pointer if the cast is not valid.

6. To ensure that the casting won't throw a null pointer, it's a good practice to use virtual functions in the base class and overriding them in a leaf class. For example, given that **ChildClass** is derived from **BaseClass** for our particular example below

   ```
   virtual bool BaseClass::is_child() { returns false; }
   bool ChildClass::is_child() override { returns true; }
   ```

   The keyword **virtual** allows run-time type checkings available. More concretely,

   ```
   BaseClass *ptr = new ChildClass();
   return ptr->is_child();
   ```

   this code snippet returns **true**. However if you remove **virtual** from the function declaration, the code snippet returns **false**. What a dark magic! isn't it? just try it.

7. There is also another way to verify the casting operation using typeid operators. Even though it seems like a syntacic sugar like Java's **instanceof** and Python's **isinstance** functions, C++ is statically-typed language running bare-metal on CPU, **typeid** is indeed a work-around solution. But it's up to you to pick any of those methods.

6 Appendix

Here is header file **type.h** containing prototypes for nullary (3.1), unary (3.2), and binary (3.3) operators. It is critical to know which attribute to use in leaf classes. For example in

Addition operator (3.3.1), since it's derived from binary operator (3.3), you should you **lhs_** and **rhs_**.

```
// A symbolic constant / nullary operator
class __nullary_op_t : public __expr_t {
public:
    __nullary_op_t() {}
    virtual ~__nullary_op_t();

    // overriden methods for instance check via run-time polymorphism
    bool is_nullary() const override;

    // base methods for instance check via run-time polymorphism
    virtual bool is_con() const;
    virtual bool is_var() const;
};

// A symbolic unary operator
class __unary_op_t : public __expr_t {
public:
    __unary_op_t(const __expr_t* op) : operand(op) {}
    virtual ~__unary_op_t();

    // overriden methods for instance check via run-time polymorphism
    bool is_unary() const override;

    // base methods for instance check via run-time polymorphism
    virtual bool is_neg() const;
    virtual bool is_exp() const;

protected:
    const __expr_t *operand;
};

// A symbolic binary operator
class __binary_op_t : public __expr_t {
public:
    __binary_op_t(const __expr_t* lhs, const __expr_t* rhs)
            : lhs_(lhs), rhs_(rhs) {}
    virtual ~__binary_op_t();

    // overriden methods for instance check via run-time polymorphism
    bool is_binary() const override;

    // base methods for instance check via run-time polymorphism
    virtual bool is_add() const;
    virtual bool is_mul() const;

protected:
    const __expr_t *lhs_, *rhs_;
};
```