Оглавление

1 Введение

При решении задач математического моделирования процессов и объектов часто очень практично использовать различные библиотеки и инструменты для упрощения вычислений. Одной из таких важных и востребованных областей является символьная математика, которая дает пользователям возможность работать с математическими выражениями в их символической форме, а не только с числовыми значениями. Такие вычисления становятся особо полезными в контексте научных исследований, инженерных разработок, а также при автоматизации различных аналитических задах [8].

2 Теоретическая часть

2.1 Символьные вычисления

Символьные вычисления или символическая математика относятся к вычислениям с математическими выражениями в их аналитической форме, то есть как с символами (переменных, констант, операторов и функций). В отличие от численных вычислений, которые производят операции с численными значениями, символьные вычисления оперируют математическими выражениями в алгебраической форме.

Основные концепции символьных вычислений

- Символьные представления: математические выражения хранятся в памяти компьютера либо в виде стека, либо в виде деревьев (Рисунок 1), где узлы представляют собой операторы, а листья - константы или переменные.
- Алгоритмы и методы: ключевыми являются алгоритмы для упрощения выражений, дифференцирования и вычисления значений.
- Системы управления памятью: важно эффективно управлять памятью для хранения сложных математических структур, особенно для больших выражений.

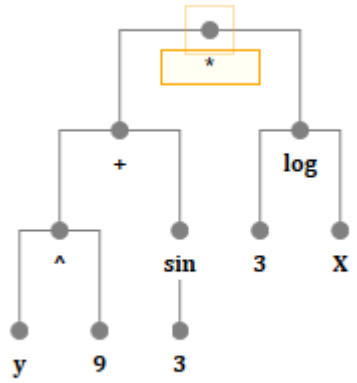


Рисунок 1 — Пример дерева выражения

Основные сценарии использования библиотеки символьной алгебры [7]:

- Вычислять математические выражения
- Подстановка символов или числовых значений
- Символически работать с переменными
- Проводить символьное дифференцирование выражений
- Выполнять посильные упрощения выражения в случае, если они могут помочь достичь более точный результат

Основу символьной части системы должно составлять упрощение выражений [4]

Таблица упрощений

$a + 0 = a$	$0 + a = a$
$a – a = 0$	$-a + a = 0$
$a * 0 = 0$	$0 * a = 0$
$a * 1 = a$	$1 * a = a$
$a^0 = 1$	$1^n = 1$
$e^{\ln(x)} = x$	$\ln(1) = 0$

2.2 Символьное дифференцирование

Дифференцирование - это процесс нахождения производной функции, который является фундаментальным понятием в математическом анализе. Производная функции представляет собой скорость изменения значения этой функции при малых изменениях её аргумента.

Дифференцирование является алгоритмической процедурой, так как опирается на знание производных элементарных функций и следующих правил:

Правила дифференцирования

$$(u+v)' = u'+v'$$

$$(cu)' = cu',\ c = const$$

$$(u-v)' = u'-v'$$

$$(uv)' = u'v + uv'$$

$$\left(\frac{u}{v}\right)' = \frac{u'v - uv'}{u^2}$$

$$(f(u(x)))'_x = f'_u(u(x)) \cdot u'(x)$$

Таблица производных элементарных функций

1. $c' = 0$	11. $(\ln x)' = \frac{1}{x}$
2. $(x^\alpha)' = \alpha x^{\alpha-1},\ \alpha \in \mathbf{R}^1$	12. $(\log_a x)' = \frac{1}{x \ln a}$
3. $\left(\frac{1}{x}\right)' = -\frac{1}{x^2}$	13. $(\arcsin x)' = \frac{1}{\sqrt{1-x^2}}$
4. $(\sqrt{x})' = \frac{1}{2\sqrt{x}}$	14. $(\arccos x)' = -\frac{1}{\sqrt{1-x^2}}$
5. $(\sin x)' = \cos x$	15. $(\operatorname{arctg} x)' = \frac{1}{1+x^2}$
6. $(\cos x)' = -\sin x$	16. $(\operatorname{arcctg} x)' = -\frac{1}{1+x^2}$
7. $(\operatorname{tg} x)' = \frac{1}{\cos^2 x}$	17. $(\operatorname{sh} x)' = \operatorname{ch} x$
8. $(\operatorname{ctg} x)' = -\frac{1}{\sin^2 x}$	18. $(\operatorname{ch} x)' = \operatorname{sh} x$
9. $(e^x)' = e^x$	19. $(\operatorname{th} x)' = \frac{1}{\operatorname{ch}^2 x}$
10. $(a^x)' = a^x \ln a,\ a > 0$	20. $(\operatorname{cth} x)' = -\frac{1}{\operatorname{sh}^2 x}$

Фактически, настоящей проблемой при дифференцировании является упрощение результата, так как если его не упрощать, то производная от 2x + 1 формально выдается в виде 0*x+2*1+0

В математике существует три основных подхода к дифференцированию: символьное, численное и автоматическое.

Символьное дифференцирование основано на аналитическом вычислении производных. Производная функции представляется и вычисляется в виде нового аналитического выражения.

Преимущества:

- Открывает возможность упрощения выражений для последующих вычислений.
- Обеспечивает точные аналитические производные без ошибок округления.

Недостатки:

- Может быть вычислительно затратным для сложных выражений.
- Производная может иметь громоздкое и сложное выражение.

В других случаях можно применять подходы численного и автоматического дифференцирования, но не в нашем случае, поэтому они рассматриваться в данной работе не будут.

2.3 Обратная польская нотация

Традиционную форму записи математических выражений, когда оператор находится между операндами, называют инфиксной. Она хорошо подходит для "бумажных" вычислений и интуитивно понятна любому человеку, но для вычислений на компьютере гораздо более удобной на практике оказывается обратная польская нотация.

Обратная польская нотация (ОПН), также известная как постфиксная нотация, представляет математические выражения в форме, при которой операнды предшествуют операторам. Это контрастирует с инфиксной нотацией, в которой операторы располагаются между операндами.

Пример:

Инфиксное выражение: (3 + 4) * 5

Постфиксная запись: 3 4 + 5 *

Из явных преимуществ ОПН можно выделить отсутствие скобок, простоту вычислений, эффективность выполнения. Но в данной работе мы будем использовать ОПН не для вычислений на стековой машине, а как промежуточное представление для более эффективного и интуитивно понятного построения абстрактного синтаксического дерева (дерева операций), речь о котором пойдет позднее.

2.4 Алгоритм сортировочной станции

Вместо построения Parse Tree гораздо легче и эффективнее будет переводить выражение из инфиксной формы в постфиксную запись. Для этого будем использовать “алгоритм сортировочной станции” (Shunting yard). Он был разработан Эдсгером Дейкстрой и часто применяется в задачах, связанных с разбором математических выражений.

Сложность по времени: O(n)

Сложность по памяти: O(n)

Алгоритм состоит из 4 основных этапов**:**

1. Алгоритм анализирует выражение слева направо.
2. Если он встречает операнд, он немедленно помещает его в очередь результатов.

3. Если алгоритм встречает оператор, есть несколько вариантов:
    a. Если стек операторов пуст, алгоритм помещает входящий оператор в стек.
    b. Если входящий оператор имеет более высокий приоритет, чем тот оператор, что в настоящее время находится на вершине стека, входящий оператор помещается на вершину стека.
    c. Если входящий оператор имеет такой же приоритет, верхний оператор извлекается из стека и выводится в очередь, а входящий оператор помещается в стек.
    d. Если приоритет входящего оператора ниже, верхний оператор извлекается из стека и выводится в очередь, после чего входящий оператор сравнивается с новой вершиной стека.
4. Когда все выражение будет проанализировано, все оставшиеся токены перекладываются из стека операторов.

2.5 Абстрактное синтаксическое дерево

2.5.1 Определение AST

Абстрактное синтаксическое дерево (AST) является ключевым компонентом для представления структуры программного кода в современных интерпретаторах и компиляторах. Оно используется для трансляции исходного кода в форму, более удобную для анализа и преобразований. В контексте символьных вычислений AST используется для представления и манипуляции математическими выражениями. На каждом узле дерева находится элемент выражения, а структура дерева определяет вложенность и порядок операций.

AST представляет собой дерево, где каждый узел соответствует элементу синтаксиса исходного языка. В отличие от конкретного синтаксического дерева (CST), AST абстрагируется от некоторых деталей синтаксиса, фокусируясь только на значимых для анализа компонентах. Другими словами «абстрактное» означает,

что синтаксис не описывает, например, группирующие скобки, лишние пробелы и т. д.

AST — это не бинарное дерево. В зависимости от “арности” операции в исходном выражении, узел дерева может иметь разное количество детей.

2.5.2 Структура AST

Структура AST для математического выражения обычно включает в себя несколько типов узлов:

- Промежуточные узлы:
  - Узлы операторов
  - Узлы функций

- Листовые узлы:
  - Константы (pi, e)
  - Числа
  - Переменные (x, y и т. п.)
  - Функции без аргументов (генератор случайных чисел)

Структура дерева должна быть иерархической, отражающей верный порядок вычислений. Например, выражение x + 2 * sin(y) будет иметь вид, как на рисунке 2.

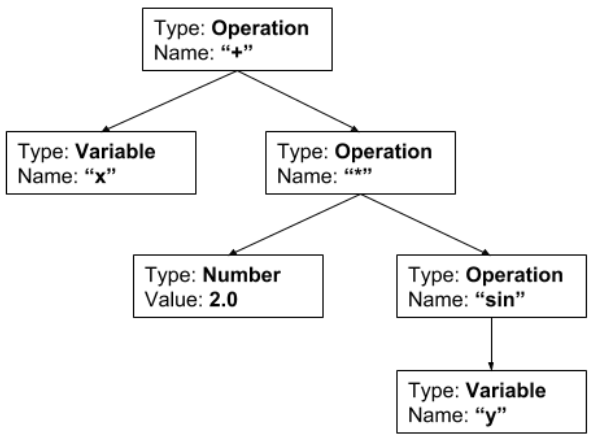


Рисунок 2 — Пример AST

2.5.3 Операции над AST

Многие древовидные структуры данных, например бинарные деревья поиска, поддерживают операции поиска, вставки и удаления. Но с AST подобные операции на практике не имеют смысла, так как оно может измениться только по одной причине — появилось что-то новое в исходном выражении. Такое изменение требует не вносить правки в старое дерево, а строить новое, поэтому единственной операцией над AST, которую необходимо будет реализовать, будет построение.

2.5.4 Построение AST

Абстрактные синтаксические деревья в основном применяются во фронтендах компиляторов, поэтому и методы построения заточены под синтаксический анализ программного кода. В данной работе они рассматриваться не будут, так как AST математического выражения гораздо легче строить по обратной польской нотации, где отсутствуют скобки и соответственно все приоритеты уже расставлены.

Данный алгоритм почти полностью повторяет алгоритм вычисления стековой машиной выражения в обратной польской нотации. Будем пробегаться по выражению, операнды кладем в стек. Если встречаем операцию, то достаем из стека нужное количество операндов и подвязываем их как детей текущего узла, который в свою очередь тоже кладем в стек. Таким образом, после окончания прохода в стеке будет лежать корень AST. Пример разбора выражения с преобразованием его структуры из линейной в древовидную представлен на рисунке 3.

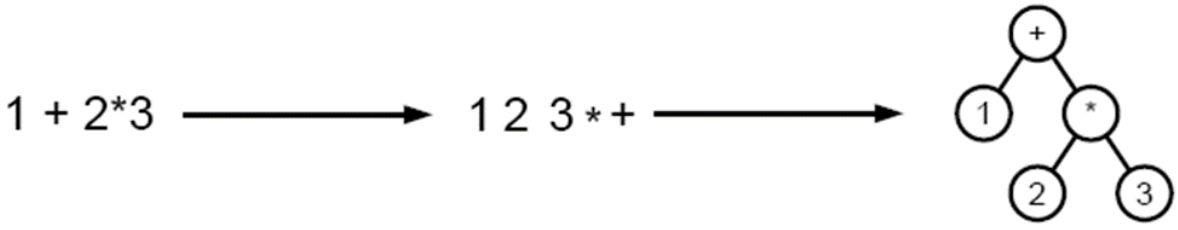


Рисунок 3 — Процесс построения AST из выражения

2.5.5 Обходы AST

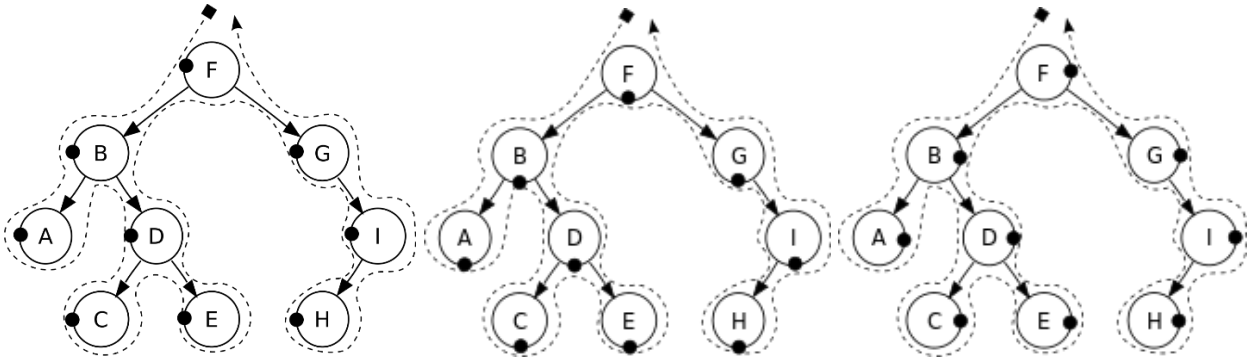
После построения AST для вычисления выражения или преобразования его в другую форму необходимо выполнить обход дерева.

Дерево — это связный ациклический граф, поэтому для него справедливы и классические обходы (обход в ширину и в глубину).

В абстрактном синтаксическом дереве обход в ширину не имеет практического смысла. Обход в глубину же подразделяется на (классификация справедлива для любых деревьев):

- Прямой обход (pre-order). Обрабатывается сначала корень, затем левое и правое поддеревья. В текущем контексте данный обход даст представление математического выражения в префиксной нотации
- Симметричный (in-order). Сначала левое поддерево, затем корень, потом правое поддерево. На выходе мы получим математическое выражение в инфиксной форме.
- Обратный (post-order). Сначала левое и правое поддеревья, затем корень. На выходе мы получим математическое выражение в обратной польской нотации.

Вариации обхода в глубину:



Прямой обход	Симметричный обход	Обратный обход
F, B, A, D, C, E, G, I, H	A, B, C, D, E, F, G, H, I	A, C, E, D, B, H, I, G, F

3 Практическая часть

3.1 Архитектура библиотеки

Ядром библиотеки является реализация абстрактного синтаксического дерева, вся логика вычислений, упрощений, дифференцирования реализована в соответствующих классах узлов AST.

Структура библиотеки изображена на рисунке 4. Можно выделить следующие модули:

- Утилиты. Синтаксический анализатор, конвертер из инфиксной формы в обратную польскую нотацию, класс-помощник.
- Ядро. Реализация узлов абстрактного синтаксического дерева.
- Обертка над AST. Символ, число, выражение.

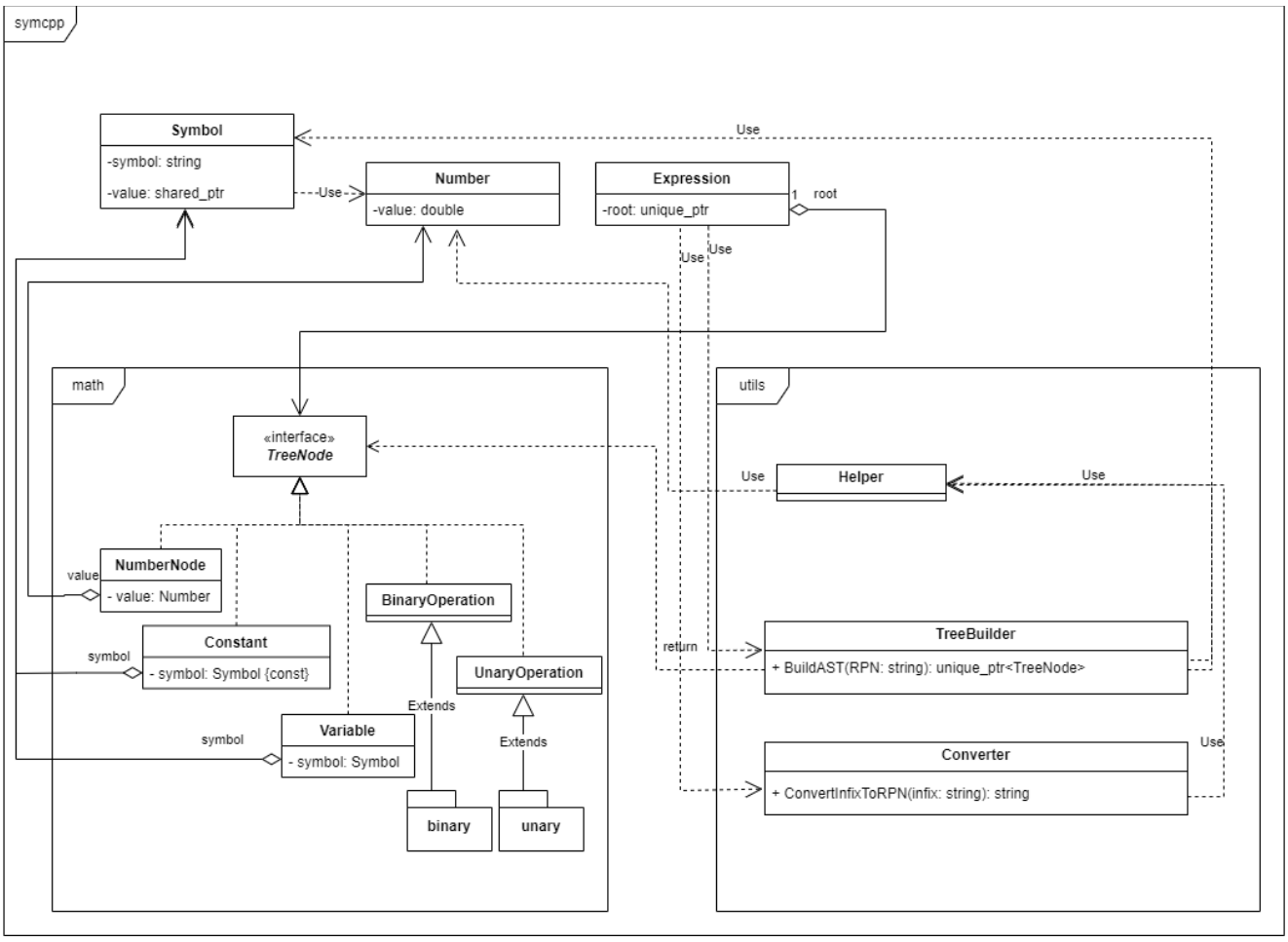


Рисунок 4 — Диаграмма классов библиотеки

3.2 Интерфейс библиотеки

Для взаимодействия с библиотекой предусмотрены два подхода (обертки над AST):

- Интерактивный. Выражение приходит только во время выполнения программы в виде строки, преобразуется в ОПН с помощью алгоритма сортировочной станции, затем синтаксический анализатор строит AST.

- Выражение известно на этапе компиляции и задано не в виде строки, а в виде программного кода. Разбор выражения будет производить компилятор, в частности он в соответствии со своими правилами будет обрабатывать скобки, старшинство операций и так далее. Библиотека перегружает стандартные арифметические операторы, для операций, не имеющих стандартного обозначения в C++, вызываются функции. Все они возвращают указатели на соответствующие узлы AST. Так, постепенно и строится дерево.

3.3 Реализация алгоритмов

3.3.1 Алгоритм сортировочной станции

Алгоритм перевода выражения из инфиксной формы в обратную польскую нотацию реализован в классе Converter (utils/converter/converter.h)

```
std::string Converter::ConvertInfixToRPN(std::string_view infix_expression) {
  std::stack<std::string_view> operators;
  std::stringstream postfix;

  for (std::size_t i = 0; i < infix_expression.size(); ++i) {
    const char& symbol = infix_expression[i];

    if (IsPrefixFunction(i, infix_expression)) {
      auto operation = *ParseFunction(i, infix_expression);
      operators.push(operation);
      i += operation.size() - 1;
    } else if (Helper::IsOperandPart(symbol)) {
      postfix << (symbol == constants::Labels::kDecimalComma ? constants::Labels::kDecimalPoint : sym-
bol);

      if (IsEndOfOperand(i, infix_expression)) {
        postfix << ' ';
      }
    } else if (symbol == constants::Labels::kOpenParenChar) {
      operators.push(infix_expression.substr(i, 1));
    } else if (symbol == constants::Labels::kEndParenChar) {
      while (!operators.empty() && operators.top() != constants::Labels::kOpenParen) {
        postfix << operators.top() << " ";
        operators.pop();
      }
```

```
      if (operators.empty()) {
        throw std::runtime_error(constants::ExceptionMessage::kUnbalancedBracket.data());
      }

      operators.pop();
    } else if (Helper::IsOperator(infix_expression.substr(i, 1))) {
      if (IsUnary(i, infix_expression)) {
        if (infix_expression.substr(i, 1) == constants::Labels::kMinus) {
          operators.emplace(constants::Labels::kUnaryMinus);
        }
      } else {
        while (!operators.empty() && operators.top() != constants::Labels::kOpenParen &&
               (Helper::ComparePriorities(operators.top(), infix_expression.substr(i, 1)) ||
                IsPrefixFunction(i, infix_expression))) {
          postfix << operators.top() << " ";
          operators.pop();
        }

        operators.emplace(infix_expression.substr(i, 1));
      }
    }
  }

  while (!operators.empty()) {
    if (operators.top() == constants::Labels::kOpenParen || operators.top() == constants::Labels::kEnd-
Paren) {
      throw std::runtime_error(constants::ExceptionMessage::kUnbalancedBracket.data());
    }

    postfix << operators.top() << " ";
    operators.pop();
  }

  return postfix.str();
}
```

Используются следующие приоритеты операторов

```
const std::unordered_map<Operations, int> operations_to_priority{{Operations::TANGENT, 6},
                                                                 {Operations::COS, 6},
                                                                 {Operations::SIN, 6},
                                                                 {Operations::SQRT, 6},
                                                                 {Operations::NATURAL_LOGARITHM, 6},
                                                                 {Operations::EXPONENTIATION, 5},
                                                                 {Operations::UNARY_MINUS, 4},
                                                                 {Operations::DIVISION, 4},
                                                                 {Operations::MULTIPLICATION, 3},
                                                                 {Operations::SUBTRACTION, 2},
                                                                 {Operations::ADDITION, 1}};
```

3.3.2 Построение AST из ОПН

Алгоритм реализован в классе TreeBuilder (utils/parser/ast_builder.h)

```
std::unique_ptr<math::TreeNode> TreeBuilder::BuildAST(std::string_view rpn_expression) {
  Reset();

  if (rpn_expression.empty()) {
    throw std::invalid_argument(constants::ExceptionMessage::kEmptyExpression.data());
  }

  std::istringstream istream{rpn_expression.data()};
  std::string input;

  while (istream >> input) {
    if (utils::Helper::IsOperator(input)) {
      if (nodes_.empty()) {
        throw std::invalid_argument(constants::ExceptionMessage::kNoOperands.data());
      }

      if (auto operation = utils::Helper::ParseOperation(input)) {
        AddOperation(*operation);
      } else {
        throw std::invalid_argument(constants::ExceptionMessage::kWrongFormat.data() + input);
```

```
      }
    } else {
      AddOperand(input);
    }
  }

  return GetOperand();
}
```

Использование

```
Symbol x('x');
Expression exp("23 + x + ln(e) + 0 + 1 * sin(pi)", {x});

std::cout << exp << std::endl; // 23 + x + 1 + sin(3.141593) Произошло упрощение
```

3.3.3 Абстрактный класс узла AST

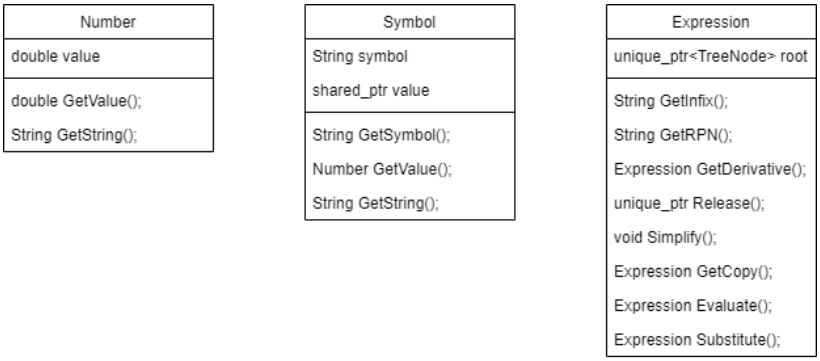
Класс TreeNode (abstract-syntax-tree/node.h). Он предоставляет весь функционал по работе с деревом: получение строкового представления выражения в разных нотациях, вычисление выражения, получение аналитической производной, подстановка переменных, упрощение и копирование.

У данного класса есть наследники: Variable, Constant, Number, UnaryOperation, BinaryOperation. Полная иерархия наследования узлов абстрактного синтаксического дерева приведена на UML-диаграмме в приложении.

```
class TreeNode {
 public:
  virtual bool IsContainVariable(const Symbol& variable) = 0;
  virtual constants::Expressions GetType() = 0;

  virtual std::string GetInfix(int previous_priority) = 0;
  virtual std::string GetRPN() = 0;

  virtual std::unique_ptr<TreeNode> Evaluate() = 0;
  virtual std::unique_ptr<TreeNode> GetDerivative(const Symbol& d) = 0;
  virtual std::unique_ptr<TreeNode> Substitute(
      const std::unordered_map<Symbol, std::unique_ptr<TreeNode>, SymbolHash>& varia-
ble_to_value) = 0;

  virtual std::unique_ptr<TreeNode> Simplify() = 0;
  virtual std::unique_ptr<TreeNode> Clone() = 0;

  virtual ~TreeNode() = default;
};
```

3.3.4 Построение AST компилятором

Для реализации интерфейса библиотеки для C++ были написаны обертки над числом, символом и самим деревом.



Чтобы проводить символьные вычисления более удобным способом, приближенным к выполнению стандартных арифметических операций на C++, необходимо еще перегрузить операторы языка и создать функции.

```
Expression operator+(Expression lhs, Expression rhs);
Expression operator-(Expression lhs, Expression rhs);
Expression operator*(Expression lhs, Expression rhs);
Expression operator/(Expression lhs, Expression rhs);

Expression operator-(Expression number);
Expression operator+(Expression number);

Expression Log(Expression argument);
Expression Sin(Expression argument);
Expression Cos(Expression argument);
Expression Tan(Expression argument);
Expression Sqrt(Expression argument);
Expression Pow(Expression base, Expression power);
```

В приведенном ниже примере мы задаем две переменные x и y, в момент инициализации объекта класса Symbol в динамической памяти создается ячейка со значением этой переменной, теперь все узлы Variable в дереве с таким же символом будут ссылаться на нее. На второй строке создается алиас z для переменной x, число 17. Чтобы создать константу, достаточно дописать квалификатор const (методы Symbol перегружены по константности)

```
Symbol x('x'), y('y');
Symbol& z(x);  // alias for x
Number num = 17;
const Symbol pi("pi", std::numbers::pi);
```

Благодаря перегрузкам операторов и функциям в первой строке создается Expression, хранящий указатель на корень дерева. Дальше переменным присваиваются значения, мы обращаемся к локальной переменной, но ее изменение затрагивает буфер на куче, таким образом, одним действием удается обновить за O(1)

значения всех переменных x в дереве. Чтобы сбросить значение достаточно вызвать метод Reset().

```
auto exp = Pow(x, 2) * 13 + Sin(pi * y);  // exp = x ^ 2 * 13 + sin(pi * y)

x = 3;         // exp = 3 ^ 2 * 13 + sin(pi * y)
y = num / 2;   // exp = 3 ^ 2 * 13 + sin(pi * 8.5)

x.Reset();  // exp = x ^ 2 * 13 + sin(pi * 8.5)
y.Reset();  // exp = x ^ 2 * 13 + sin(pi * y)
```

3.3.5 Вычисление результата

Чтобы получить численный результат вычисления выражения, можно пойти двумя путями. Либо приравнять выражение к Number, либо вручную вызвать у Expression метод Evaluate().

Для примера, приведенного выше.

```
Number result = exp;  // result = 118
Number result_2 = Evaluate(exp, {{x, 14}, {y, 17}});  // result_2 = 7.843064
```

В этот момент произойдет симметричный обход дерева. Рекурсивно вызовутся методы Evaluate для левого и правого поддеревьев. После этого посчитается результат текущего узла. Базовыми случаями для рекурсии будут три узла: число, переменная (нужно задать значение) и константа (значение должно быть задано при объявлении).

Если не для всех переменных заданы значения, ошибки не будет, выражение просто посчитается не полностью и максимально попытается упроститься. Исключение вылетит только в случае, если намеренно будет проведен каст к Number.

Пример вычисления для умножения:

```
std::unique_ptr<TreeNode> Multiplication::Evaluate() {
  auto left_result = left_argument_->Evaluate();
  auto right_result = right_argument_->Evaluate();

  if (auto left = GetNumber(left_result)) {
    if (auto right = GetNumber(right_result)) {
      return std::make_unique<NumberNode>(*left * *right);
    }
  }

  return std::make_unique<Multiplication>(std::move(left_result), std::move(right_result));
}
```

Временная сложность идентична обходу в глубину: O(n)

Пространственная сложность: O(n) (строится новое дерево)

3.3.6 Дифференцирование

Взятие производной также производится симметричным обходом в глубину. Каждый узел знает, каким образом он должен дифференцироваться. Базовыми случаями вновь являются листовые узлы: переменная = 1, константы и числа = 0. Для всех промежуточных узлов заданы правила дифференцирования, сначала рекурсивно вызывается метод GetDerivative() у детей, затем конструируется нужная производная исходя из контекста (тип текущие узла). Вторым параметром функция Diff принимает переменную, по которой будет проводится дифференцирование.

```
auto derivative_of_func_x = Diff(exp, x);  // f`x(x,y) = 2 * x * 13
auto derivative_of_func_y = Diff(exp, y);  // f`y(x,y) = cos(pi * y) * pi
```

Пример взятия производной синуса

```
std::unique_ptr<TreeNode> SinNode::GetDerivative(const Symbol& d) {
  return std::make_unique<Multiplication>(std::make_unique<CosNode>(argument_-
>Clone()), argument_->GetDerivative(d));
}
```

Временная сложность: O(n)

Пространственная сложность: O(n) (строится новое дерево)

3.3.7 Упрощение

Каждый узел знает, как он может быть упрощен. Например, умножение может определить, что один из потомков равен нулю и стоит вернуть 0 на уровень выше. Таблица упрощений была приведена в теоретической части.

Реализуется также симметричным обходом.

Пример для логарифма

```
std::unique_ptr<TreeNode> LogarithmNode::Simplify() {
  if (auto simplified = argument_->Simplify()) {
    argument_ = std::move(simplified);
  }

  if ((argument_->GetType() == constants::Expressions::NUMBER &&
       utils::Helper::IsEqual(*GetNumber(argument_), std::numbers::e)) ||
      (argument_->GetType() == constants::Expressions::CONSTANT && argument_->IsContainVar-
iable('e'))) {
    return std::make_unique<NumberNode>(1);
  }

  return nullptr;
}
```

Временная сложность идентична обходу в глубину: O(n)

Пространственная сложность: O(n) (в худшем случае, производится in-place)

3.3.8 Получение линейного представления выражения (строки)

Для получения строкового представления выражения реализованы два метода: GetRPN() и GetInfix(). Перегружен оператор вывода <<, по умолчанию использующий инфиксную нотацию.

- Получение обратной польской нотации происходит путем post-order обхода.
- Получение инфиксной нотации происходит путем in-order обхода по AST.

Во втором случае важно также верно расставлять скобки для сохранения порядка выполнения. Чтобы не выводить лишних скобок также необходимо постоянно сравнивать приоритеты операций.

Пример для деления:

```
std::string Division::GetInfix(int previous_priority) {
  bool brackets_required = previous_priority >= priority_;
  std::stringstream stream;
  stream << (brackets_required ? constants::Labels::kOpenParen : "") << left_argument_->GetInfix(prior-
ity_) << " "
         << constants::Labels::kDivision << " " << right_argument_->GetInfix(priority_)
         << (brackets_required ? constants::Labels::kEndParen : "");
  return stream.str();
}

std::string Division::GetRPN() {
  std::stringstream stream;
  stream << left_argument_->GetRPN() << " " << right_argument_->GetRPN() << " " << constants::La-
bels::kDivision;
  return stream.str();
}
```

Пример использования:

```
// Инфиксная форма
std::cout << "exp = " << exp << std::endl;
std::cout << "exp = " << Infix(exp) << std::endl;

// ОПН
std::cout << "RPN = " << RPN(exp + Log(Pow(x, y))) << std::endl;
```

Временная сложность: O(n)

Пространственная сложность: O(n)

4 Примеры использования

4.1 Базовый пример

```cpp
#include <iostream>
#include "symcpp.h"

/**
 * @brief Example from README.md
 */
int main() {
  using namespace symcpp;
  Symbol x('x'), y('y');
  Symbol& z(x);  // alias for x
  Number num = 17;

  auto exp = Pow(x, 2) * 13 + Sin(pi * y);  // exp = x ^ 2 * 13 + sin(pi * y)

  x = 3;         // exp = 3 ^ 2 * 13 + sin(pi * y)
  y = num / 2;   // exp = 3 ^ 2 * 13 + sin(pi * 8.5)

  Number result = exp;  // result = 118

  x.Reset();  // exp = x ^ 2 * 13 + sin(pi * 8.5)
  y.Reset();  // exp = x ^ 2 * 13 + sin(pi * y)

  auto derivative_of_func_x = Diff(exp, x);  // f`x(x,y) = 2 * x * 13
  auto derivative_of_func_y = Diff(exp, y);  // f`y(x,y) = cos(pi * y) * pi

  Number result_2 = Evaluate(exp, {{x, 14}, {y, 17}});  // result_2 = 7.843064

  RPN(exp + Log(Pow(x, y)))  // x 2 ^ 13 * pi y * sin + x y ^ ln +
}
```

4.2 Построение касательной

```cpp
#include <iostream>
#include "symcpp.h"

/**
 * @brief Example. Tangent builder
 * @input x coordinate
 */
int main() {
  using namespace symcpp;

  Symbol x('x');

  auto func = 12 * Sin(Log(x) * x) - Pow(x, Tan(x));
  std::cout << "f(x) = " << func << std::endl;

  auto derivative = Diff(func, x);
  std::cout << "f`(x) = " << derivative << std::endl;

  while (true) {
    int input;
    std::cout << "Enter x: ";
    std::cin >> input;

    x = input;
```

```
    auto y = Evaluate(func);
    std::cout << "f(" << x << ") = " << y << std::endl;

    auto k = Evaluate(derivative);
    auto b = Evaluate(y - k * x);

    auto tangent = k * x + b;
    x.Reset();

    std::cout << "Tangent: g(x) = " << tangent << std::endl;

    std::cout << std::endl;
  }
}
```

4.3 Интерактивный режим

Построение дерева происходит в во время выполнения программы с помощью алгоритма сортировочной станции. На этапе компиляции выражение неизвестно.

```
#include <iostream>
#include "symcpp.h"

/**
 * @brief Example of interactive calculations
 */
int main() {
  using namespace symcpp;
  std::string symbol;
  std::cout << "Enter variable symbol: ";
  std::cin >> symbol;

  Symbol var(symbol);

  std::cout << "Enter expression: ";
  std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
  std::string expr_str;
  std::getline(std::cin, expr_str);
  Expression exp(expr_str, {var});

  std::cout << "Simplified: " << exp << std::endl;
  std::cout << "RPN: " << RPN(exp) << std::endl;
  std::cout << "f`(x) = " << Diff(exp, var) << std::endl;

  std::cout << "Enter x: " << std::endl;
  std::cin >> var;

  std::cout << "f(" << var << ") = " << Evaluate(exp) << std::endl;

  return EXIT_SUCCESS;
}
```

5 Возможные улучшения

- В дальнейшем в библиотеку можно добавлять более сложные математические объекты (векторы, матрицы, комплексные числа), функции и операции.
- Библиотека соответствует принципу OCP (Open Closed Principle), поэтому добавление новых операций (например, гиперболических функций) не потребует изменять уже работающий код.
- Можно научить узлы генерировать строку в формате LaTeX.
- Можно добавить продвинутые алгоритмы упрощения выражений, поиск в дереве некоторых паттернов. Например, $\sin^2(x) + \cos^2(x) = 1$ или $\sin(x) + \sin(x) = 2*\sin(x)$.

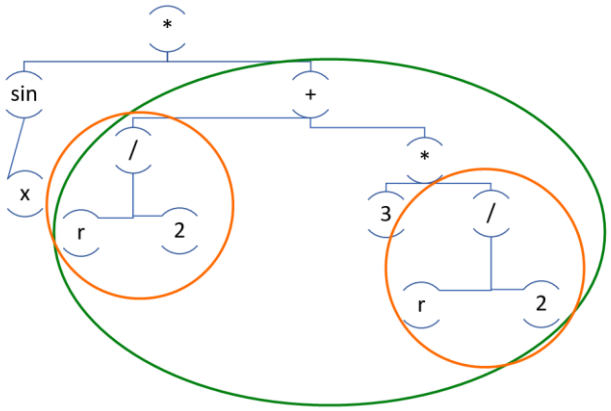


Рисунок 5 – Анализ поддеревьев

- Можно улучшить упрощение ассоциативных и коммутативных операций путем добавления n-арных узлов в структуру дерева.

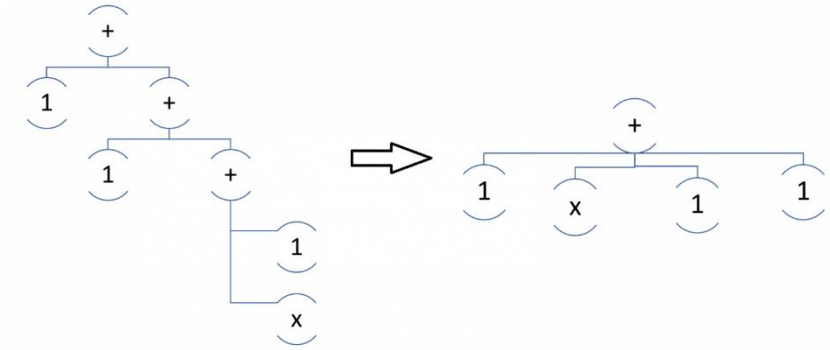


Рисунок 6 – Пример n-арного сложения

6 Заключение

В ходе выполнения курсового проекта была разработана библиотека для символьных вычислений на языке программирования C++ применительно к стандарту C++ 20. Были реализованы алгоритм сортировочной станции, синтаксический анализатор, абстрактное синтаксическое дерево для представления математического выражения, интерфейс для использования сторонними разработчиками в других проектах.

Созданная библиотека позволяет символически манипулировать математическими выражениями, дифференцировать их, вычислять и упрощать без потери точности, в отличие от численных методов. Добавлена возможность интерактивных вычислений во время выполнения программы (runtime), как в системах компьютерной алгебры.

Полученное решение спроектировано с учетом возможной кастомизации со стороны стороннего разработчика. В библиотеку можно добавлять новые операции, функции, математические объекты, почти не затрагивая уже написанный код.

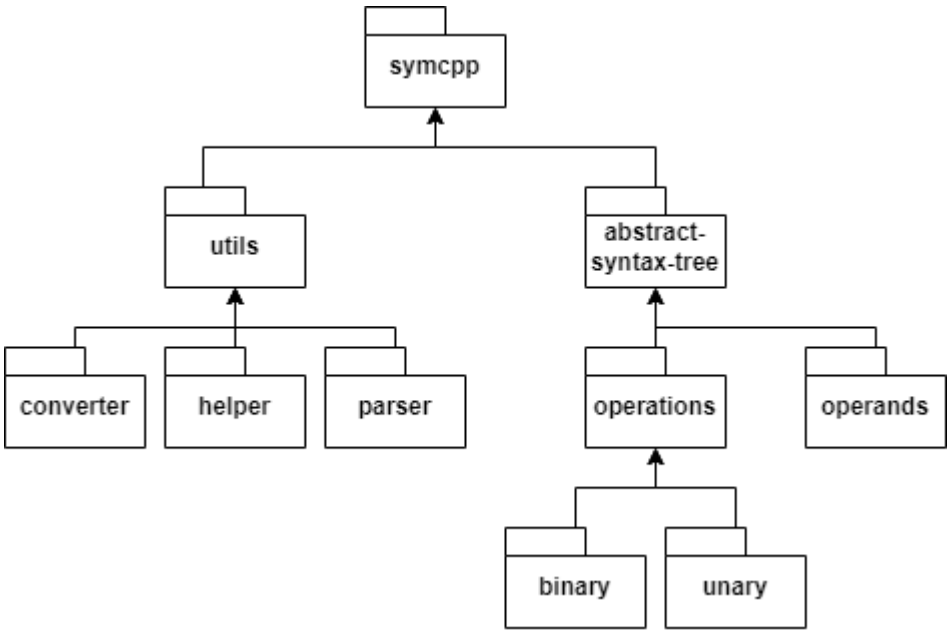
Проведен обзор уже существующих решений, теоретической части, примененной в проекте, описаны возможности библиотеки, приведены примеры использования.

7 Список литературы и ссылки

1. Задание на курсовой проект по дисциплине «Языки программирования» – Уфа: УУНИТ, 2024
2. Бьярне Страуструп «Язык программирования C++», Четвертое издание, 2011
3. Цикл статей, описывающих процесс разработки библиотеки символьной алгебры на C#: habr.com/ru/users/WhiteBlackGoose/publications/articles/
4. University of Washington Computer Science & Engineering. Implementation of symbolic algebra calculator. Abstract syntax trees: courses.cs.washington.edu/courses/cse373/17au/project1/project1-2.html
5. Vincent Reverdy «Symbolic Calculus for High-performance Computing From Scratch Using C++23»: youtu.be/lPfA4SFojao?si=FNsesd3ioOXI-tV7
6. Joel Falcou & Vincent Reverdy «EDSL Infinity Wars: Mainstreaming Symbolic Computation»: youtu.be/XH00wB_bbU4?si=iAqYY7h_jCzgzmuY
7. Тан К. Ш., Стиб В-Х., Харди Й. «Символьный C++: Введение в компьютерную алгебру с использованием объектно-ориентированного программирования». Пер. со 2-го англ. Изд. — М.: Мир, 2001. — 622 с., ил.
8. Краснов М.М. Применение символьного дифференцирования для решения ряда вычислительных задач // Препринты ИПМ им. М. В. Келдыша. 2017. № 4. 24 с. doi:10.20948/prepr-2017-4 URL: library.keldysh.ru/preprint.asp?id=2017-4
9. Christian Bauer, Alexander Frink, Richard Kreckel «Introduction to the GiNaC Framework for Symbolic Computation within the C++ Programming Language», 2002 — 12 с. www.ginac.de/csSC-0004015.pdf
10. Скотт Мейерс «Эффективный и современный C++: 42 рекомендации по использованию C++ 11 и C++14»: Пер. с англ. - М. : ООО "ИЛ. Вильяме", 2016. - 304 с.
11. Макеев Г. А. «Объектно-ориентированное программирование: с нуля к SOLID и MVC» — СПб.: БХВ-Петербург, 2024 – 272 с

8 Приложения

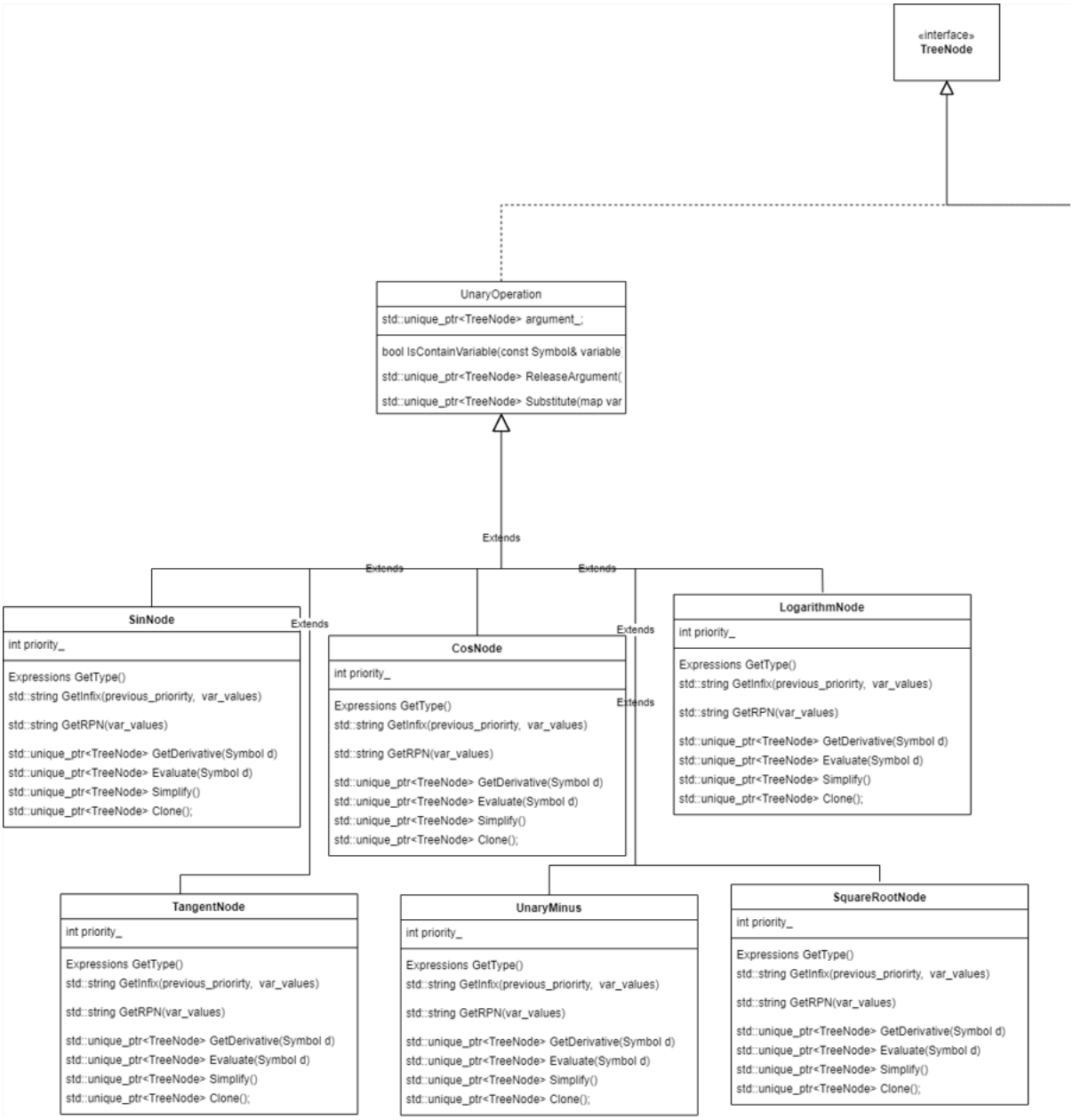
8.1 Структура каталогов библиотеки

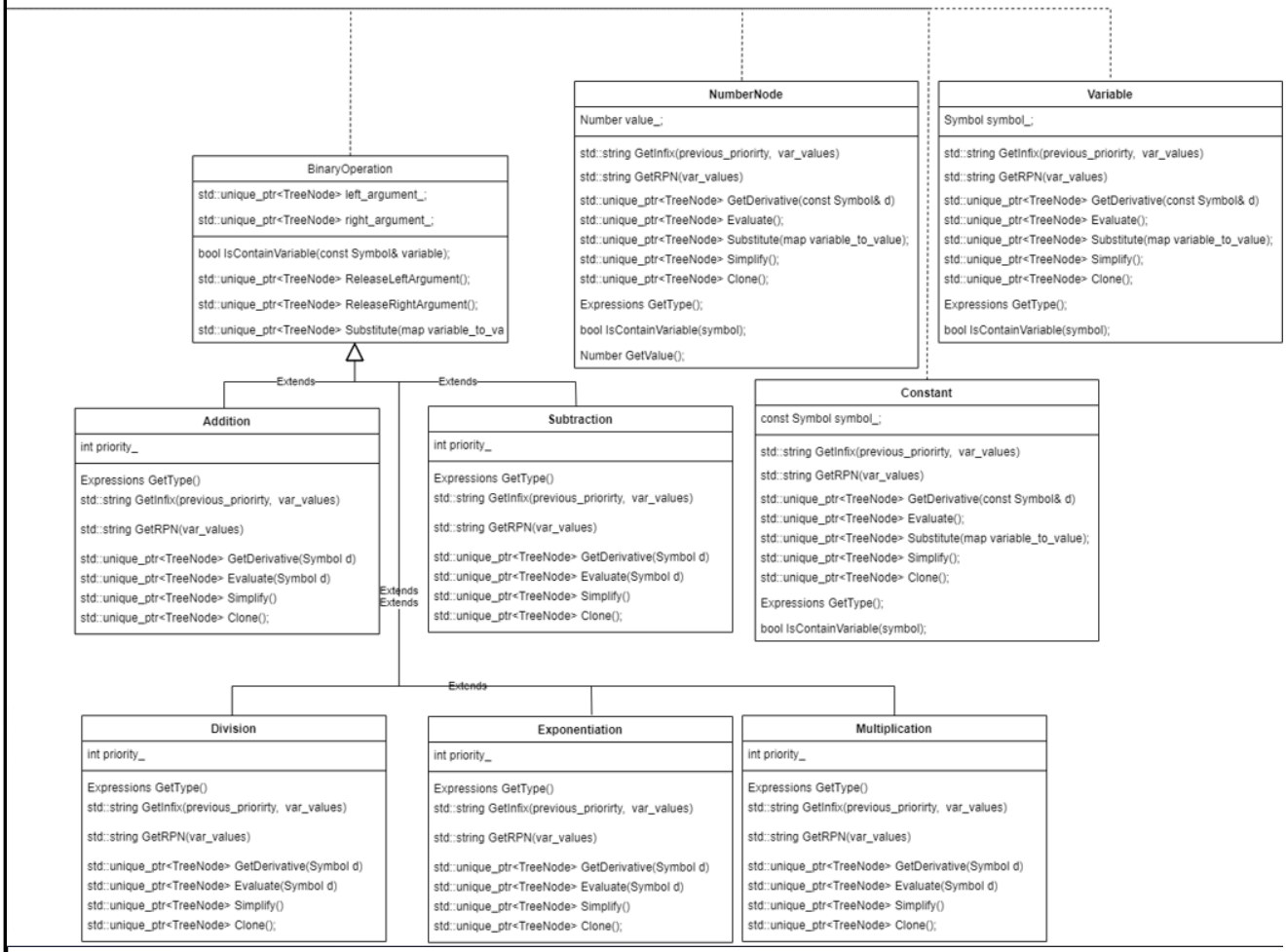


8.2 Исходный код

Исходный код библиотеки, примеры и тесты выложены в публичном Git-репозитории (https://github.com/Repin-Daniil/symbolic-math)

8.3 UML-диаграмма ядра библиотеки




BinaryOperation
std::unique_ptr<TreeNode> left_argument_;
std::unique_ptr<TreeNode> right_argument_;
bool IsContainVariable(const Symbol& variable);
std::unique_ptr<TreeNode> ReleaseLeftArgument();
std::unique_ptr<TreeNode> ReleaseRightArgument();
std::unique_ptr<TreeNode> Substitute(map variable_to_va
NumberNode
Number value_;
std::string GetInfix(previous_priorirty, var_values)
std::string GetRPN(var_values)
std::unique_ptr<TreeNode> GetDerivative(const Symbol& d)
std::unique_ptr<TreeNode> Evaluate();
std::unique_ptr<TreeNode> Substitute(map variable_to_value);
std::unique_ptr<TreeNode> Simplify();
std::unique_ptr<TreeNode> Clone();
Expressions GetType();
bool IsContainVariable(symbol);
Number GetValue();
Variable
Symbol symbol_;
std::string GetInfix(previous_priorirty, var_values)
std::string GetRPN(var_values)
std::unique_ptr<TreeNode> GetDerivative(const Symbol& d)
std::unique_ptr<TreeNode> Evaluate();
std::unique_ptr<TreeNode> Substitute(map variable_to_value);
std::unique_ptr<TreeNode> Simplify();
std::unique_ptr<TreeNode> Clone();
Expressions GetType();
bool IsContainVariable(symbol);
Extends
Extends
Addition
int priority_
Expressions GetType()
std::string GetInfix(previous_priorirty, var_values)
std::string GetRPN(var_values)
std::unique_ptr<TreeNode> GetDerivative(Symbol d)
std::unique_ptr<TreeNode> Evaluate(Symbol d)
std::unique_ptr<TreeNode> Simplify()
std::unique_ptr<TreeNode> Clone();
Subtraction
int priority_
Expressions GetType()
std::string GetInfix(previous_priorirty, var_values)
std::string GetRPN(var_values)
std::unique_ptr<TreeNode> GetDerivative(Symbol d)
std::unique_ptr<TreeNode> Evaluate(Symbol d)
std::unique_ptr<TreeNode> Simplify()
std::unique_ptr<TreeNode> Clone();
Constant
const Symbol symbol_;
std::string GetInfix(previous_priorirty, var_values)
std::string GetRPN(var_values)
std::unique_ptr<TreeNode> GetDerivative(const Symbol& d)
std::unique_ptr<TreeNode> Evaluate();
std::unique_ptr<TreeNode> Substitute(map variable_to_value);
std::unique_ptr<TreeNode> Simplify();
std::unique_ptr<TreeNode> Clone();
Expressions GetType();
bool IsContainVariable(symbol);
Extends
Extends
Extends
Division
int priority_
Expressions GetType()
std::string GetInfix(previous_priorirty, var_values)
std::string GetRPN(var_values)
std::unique_ptr<TreeNode> GetDerivative(Symbol d)
std::unique_ptr<TreeNode> Evaluate(Symbol d)
std::unique_ptr<TreeNode> Simplify()
std::unique_ptr<TreeNode> Clone();
Exponentiation
int priority_
Expressions GetType()
std::string GetInfix(previous_priorirty, var_values)
std::string GetRPN(var_values)
std::unique_ptr<TreeNode> GetDerivative(Symbol d)
std::unique_ptr<TreeNode> Evaluate(Symbol d)
std::unique_ptr<TreeNode> Simplify()
std::unique_ptr<TreeNode> Clone();
Multiplication
int priority_
Expressions GetType()
std::string GetInfix(previous_priorirty, var_values)
std::string GetRPN(var_values)
std::unique_ptr<TreeNode> GetDerivative(Symbol d)
std::unique_ptr<TreeNode> Evaluate(Symbol d)
std::unique_ptr<TreeNode> Simplify()
std::unique_ptr<TreeNode> Clone();