可编程计算器

题目描述

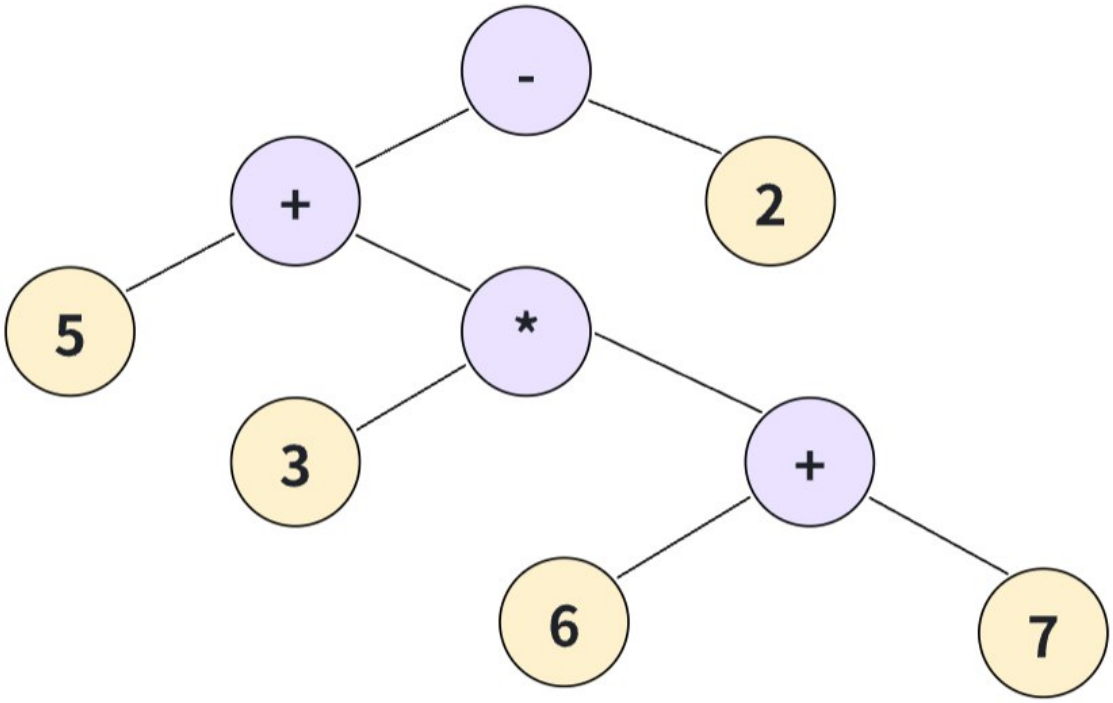
在本题目中，你需要实现一个简易的可编程计算器，可以支持四则运算表达式的计算，并可以自定义常数，以及自定义二元运算符，来表示四则运算的一种组合。**在此之前，你可以先复制文末的代码模板以方便对照。**

> 题面较长，但给足了相应提示，代码量大概在一百行左右 如果你觉得模板太过丑陋，你也可以自行设计算法，只需要保证可以正常调用即可

Part 1 解析并计算简单表达式

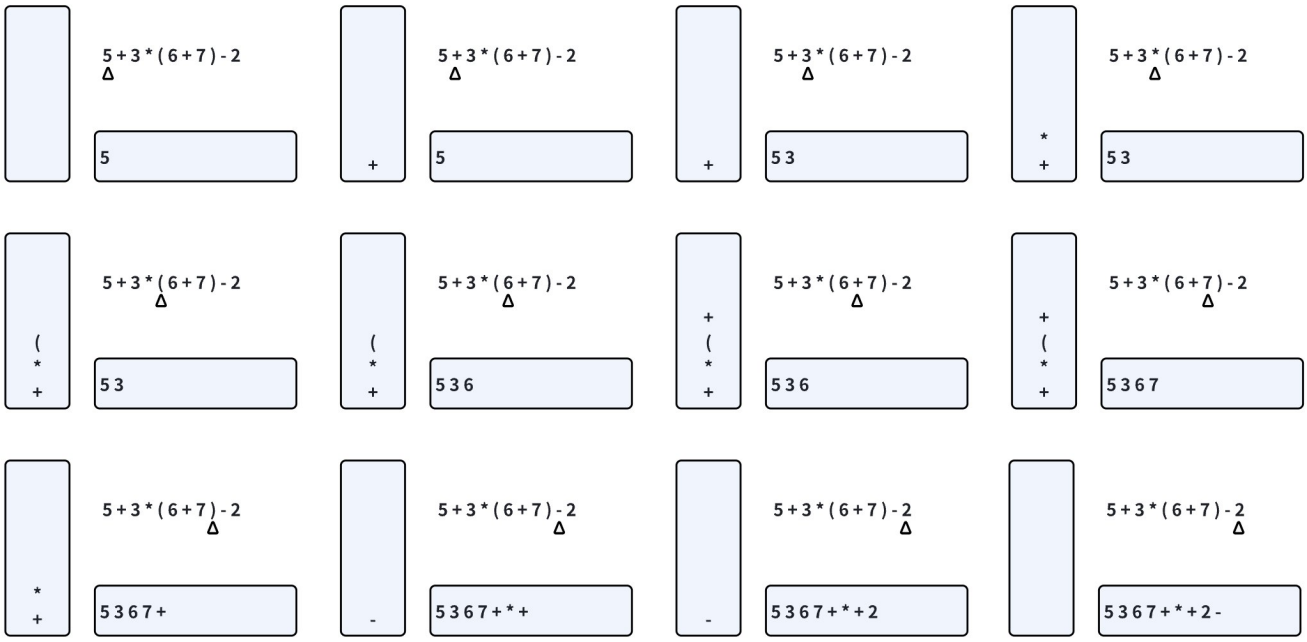
一个简单算术表达式可以用二叉树来进行表示，树的叶节点储存操作数，而非叶节点则储存操作符。例如表达式`5+3*(6+7)-2`可以用如下的二叉树来表示

> 实际上，该表达式是其对应二叉树中序遍历的结果，也被称为**中缀表达式**



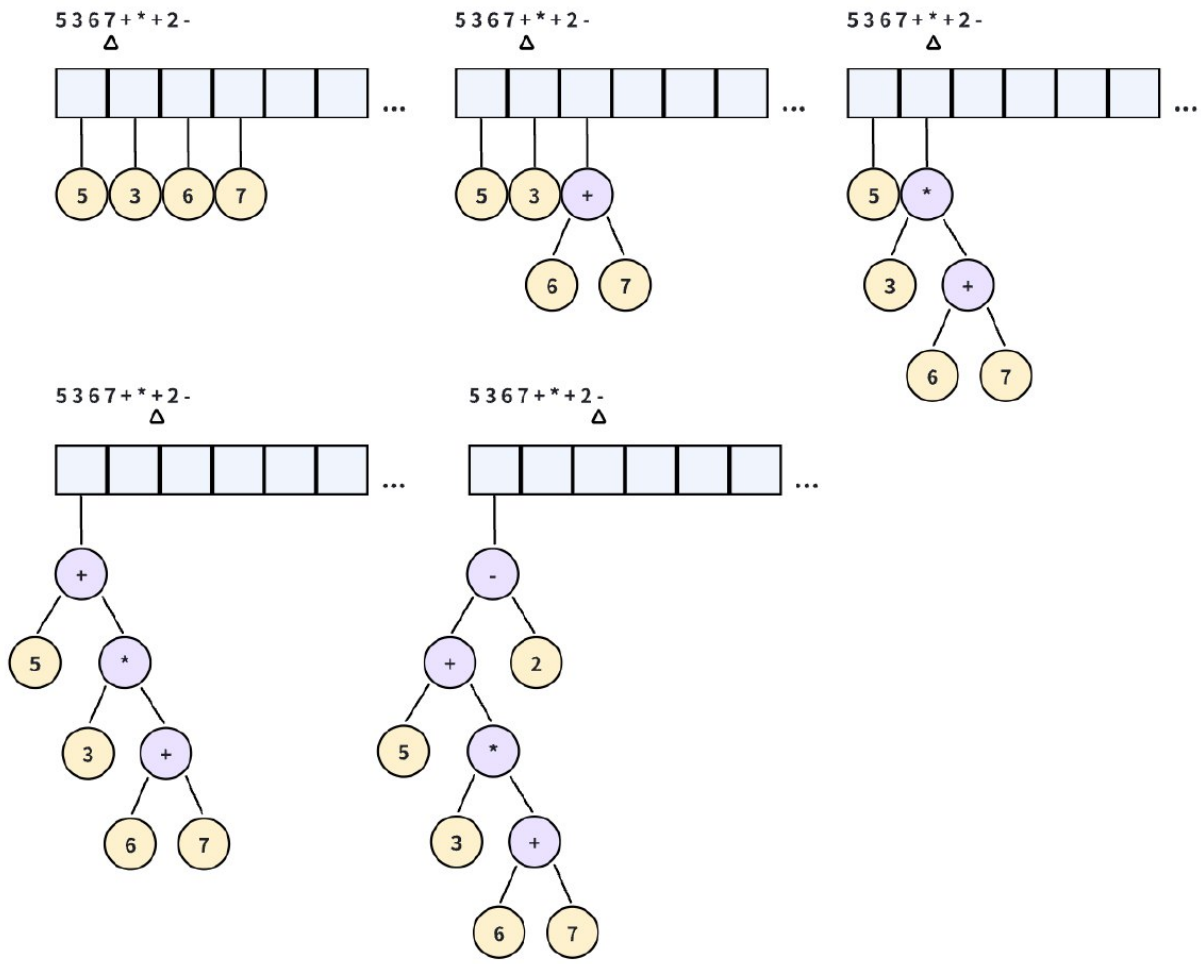
为了计算该表达式的值，我们只需要从根节点开始，递归获取左右子树的值并进行相应的运算。那该如何构建表达式的二叉树呢？其基本思想是将中缀表达式转化为后缀表达式，再用后缀表达式构造表达式二叉树。

- **后缀表达式的转换**
    1. 初始化一个操作符栈和一个数组
    2. 遍历中缀表达式的每个字符 a. 如果是操作数，则直接加入数组 b. 如果是操作符，则不断从栈中弹出操作符并添加到数组直到栈顶运算符优先级低于当前运算符，然后将当前运算符压入栈中 c. 如果是左括号，则直接压入栈 d. 如果是右括号，则不断从栈中弹出运算符并添加到数组，直到遇到对应的左括号（该左括号直接弹出丢弃）
    3. 所有字符检查完毕后，将栈中剩余的运算符依次弹出并添加到数组



- **后缀表达式构建二叉树**
    1. 初始化一个栈
    2. 遍历后缀表达式的每个字符 a. 如果是操作数，构造相应的叶节点，并将其指针压入栈中 b. 如果是操作符，则先构造节点，然后弹出栈顶两节点的指针，其所指向的节点分别作为该操作符节点的左右节点，然后将该节点的指针压入栈中
    3. 最后栈顶即为指向表达式二叉树根节点的指针

> 实际上，可以直接使用数值栈，遇到运算符时，从数值栈中弹出两个操作数，执行相应的运算，然后将结果压回栈中，这样最后可以直接得到表达式的值，但为了后续的需求，这里并不推荐这样做



在本题目的代码模板中，在TreeNode.h中定义了一个结构体TreeNode，作为树的节点，其中operate表示一个二元函数，element表示操作数或操作符。然后在Calculator类中，定义了两个成员变量以及两个成员函数

```
std::map<char, int> precedenceTable;      \\ 运算符优先级表
std::map<char, BINARY_OP> functionTable;  \\ 函数表

TreeNode* buildTree(std::vector<std::string>& tokens);

double getVal(TreeNode* node);
```

首先你需要实现Calculator的构造函数**初始化运算符优先级表以及函数表**（在此阶段只有加减乘除），我们用0~9的一个整数来表示优先级，数字越小优先级越高，其中+、-的优先级为6，*、/的优先级为5，在初始化函数表时推荐使用lambda表达式

然后根据以上的算法实现buildTree函数以及getVal函数，getVal函数可以用如下伪代码表示

```
double getVal(TreeNode* node) {
  if 是非叶节点
    return node->operate(getVal(node->left), getVal(node->right));
  if 是叶节点
```

```
        直接返回操作数对应的值
    }
```

最后在以上两个函数的基础上，你需要实现`Calculator`类的`double calculate(const std::string& expr)`函数完成简单算数表达式的计算，其中参数`expr`的约定如下：

- 仅由操作符，操作数以及`(`、`)`组成，操作符为**单个字符**（非字母非数字），操作数由**浮点数或纯字母**组成（在该阶段中仅有浮点数，在Part2中会涉及常数解析），例如：`5 + 3 * ( 6 + 7 ) - 2`
- 每个符号之间由空格隔开
- 在本题的每个阶段**输入保证表达式合法**
- **输入保证不发生溢出**

到此为止恭喜你可以拿到40%的分数（其实已经完成大部分的代码了）

Part 2 自定义常量

本阶段涉及部分如下

```
std::map<std::string, double> constantTable;  // 常数表

void registerConstant(const std::string& symbol, double val);
```

首先你需要实现`registerConstant`函数完成常数表的注册，几点说明如下

- `symbol`参数仅由**大小写字母组成**
- 注册同名常数会进行覆盖

然后你需要修改`getVal`函数完成常数的解析

到此为止恭喜你可以拿到70%的分数

Part 3 自定义运算符

在本阶段中你需要完成`Calculator`的最后一个公有成员函数，实现**二元运算符的自定义**

```
void registerOperator(const std::string& expr, int precedence);
```

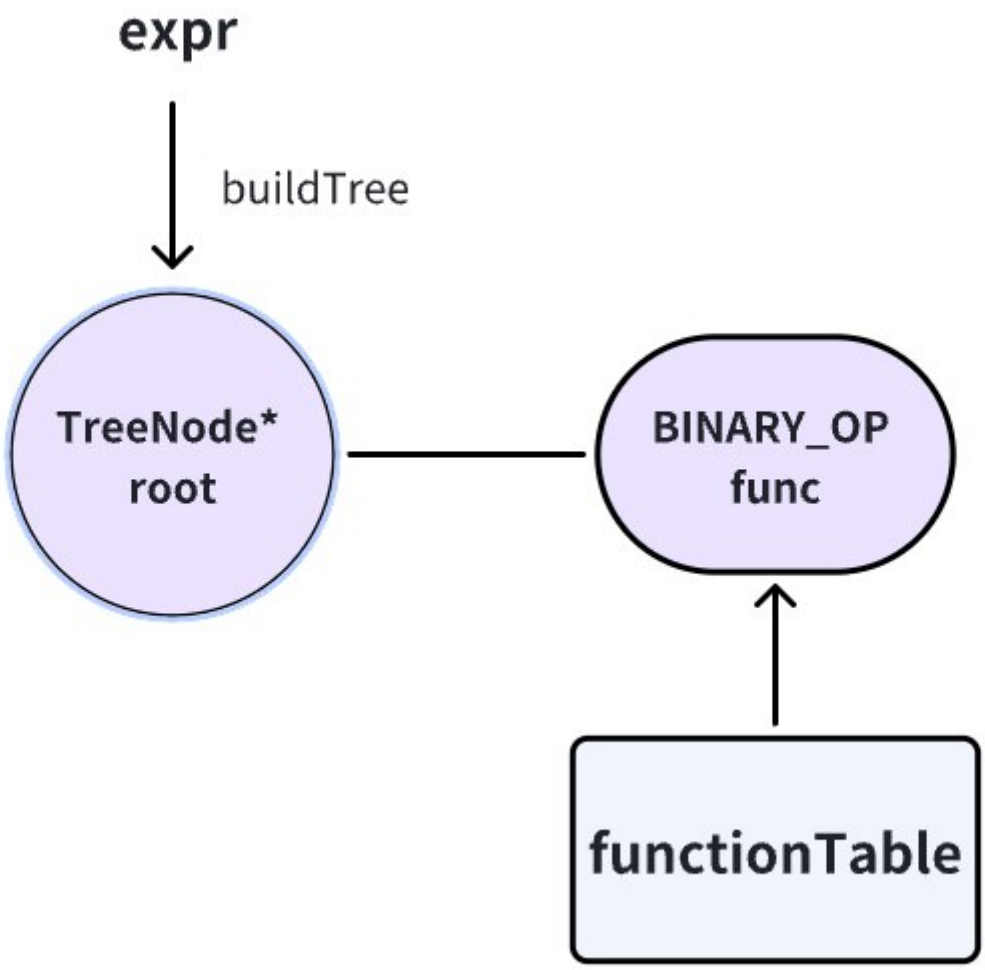
参数说明如下

- `expr`
  - 规定格式：`[左操作数名] [运算符] [右操作数名] = [表达式]`
  - 左操作数名与右操作数名均由纯字母组成，且不与已定义的常数名冲突
  - 可自定义的运算符有如下10种：`!`、`@`、`#`、`$`、`%`、`^`、`&`、`<`、`>`、`?`
  - 表达式的可能组成：左操作数、右操作数、浮点数、已定义的常数、四则运算符、已定义的运算符
  - 各符号间由空格隔开

```
// 几个例子
calculator.registerConstant("pi", 3.1415);
calculator.registerOperator("x & y = 2 * x + 3 * y", 5);
calculator.registerOperator("x ^ y = x & y * pi", 5);
// x ^ y = ( 2 * x + 3 * y ) * 3.1415
```

- `precedence`
  - `0~9`的一个整数
  - 数字越小优先级越高

一种可能的实现方式如图所示



首先你需要解析表达式构建二叉树，在这个过程中你需要用一种方式特殊**标记左右操作数**，并重载`getVal`函数，以在实际计算过程中将左右操作数的值作为参数传入

然后实现该二叉树与相应函数的绑定，这里有两种可能的方式，一种是事先将所有操作符的函数定义好，在函数体中通过一个`map`访问对应的二叉树，在操作符定义时再将该函数注册到`functionTable`中，但这样有可能导致代码的冗余，所以另一种更推荐的方式是使用**带捕获的lambda表达式**，具体参考以下链接

https://oi-wiki.org/lang/lambda/

如果你完成了以上内容，恭喜你可以拿到全部的分数了！

调用示例

Part 1 示例

```
Calculator calculator;
cout << calculator.calculate("5 + 3 * ( 6 + 7 ) - 2") << endl;
```

输出:

```
42
```

Part 2 示例

```
Calculator calculator;
calculator.registerConstant("pi", 3.1415);
cout << calculator.calculate("2 * 2 * pi");
```

输出:

```
12.566
```

Part 3 示例

```
Calculator calculator;
calculator.registerConstant("x", 2);
calculator.registerOperator("h ^ l = h * l / x", 5);
calculator.registerConstant("h", 2);
calculator.registerConstant("l", 2);
cout << calculator.calculate("h ^ l") << endl;
calculator.registerConstant("h", 3);
calculator.registerConstant("l", 3);
cout << calculator.calculate("h ^ l") << endl;
```

输出:

```

```

```
2
4.5
```

代码模板

Main.cpp留空即可，系统测试时会进行替换

TreeNode.h

```
#include <functional>
#include <string>

// 函数类型别名
typedef std::function<double(double, double)> BINARY_OP;

struct TreeNode {
  TreeNode* left{nullptr};
  TreeNode* right{nullptr};
  BINARY_OP operate{nullptr};
  std::string element;

  // 叶节点构造函数
  TreeNode(std::string e)
      : element(e) {}

  // 非叶节点构造函数
  TreeNode(BINARY_OP o, std::string e, TreeNode* l, TreeNode* r)
      : operate(o), element(e), left(l), right(r) {}
};
```

Calculator.h

```
#include <map>
#include <vector>
#include "TreeNode.h"

class Calculator {
 private:
  std::map<char, int> precedenceTable;          // 运算符优先级表
  std::map<char, BINARY_OP> functionTable;      // 函数表
  std::map<std::string, double> constantTable;  // 常数表

  TreeNode* buildTree(std::vector<std::string>& tokens);

  double getVal(TreeNode* node);

 public:
```

```
    Calculator();

    void registerOperator(const std::string& expr, int precedence);

    void registerConstant(const std::string& symbol, double val);

    double calculate(const std::string& expr);
};
```

Calculator.cpp

```
#include "Calculator.h"

using namespace std;

Calculator::Calculator() {
    // TODO: 初始化Calculator
}

void Calculator::registerOperator(const string& expr, int precedence) {
    // TODO: 注册运算符
}

void Calculator::registerConstant(const string& symbol, double val) {
    // TODO: 注册常数
}

double Calculator::calculate(const string& expr) {
    // TODO: 解析并计算表达式
    return 0;
}

TreeNode* Calculator::buildTree(vector<string>& tokens) {
    // TODO: 解析表达式并构造表达式解析树
    return nullptr;
}

double Calculator::getVal(TreeNode* node) {
    // TODO: 获取树上某一节点的值
    return 0;
}
```