软件构造实验报告八

实验名称：解释器模式编程实现

实验时间：2019.5.29

学号：E21614061

姓名：徐奕

所在院系：计算机科学与技术学院

所在专业：软件工程

【实验目的和要求】

利用解释器模式，打印并计算实现浮点数的四则运算。

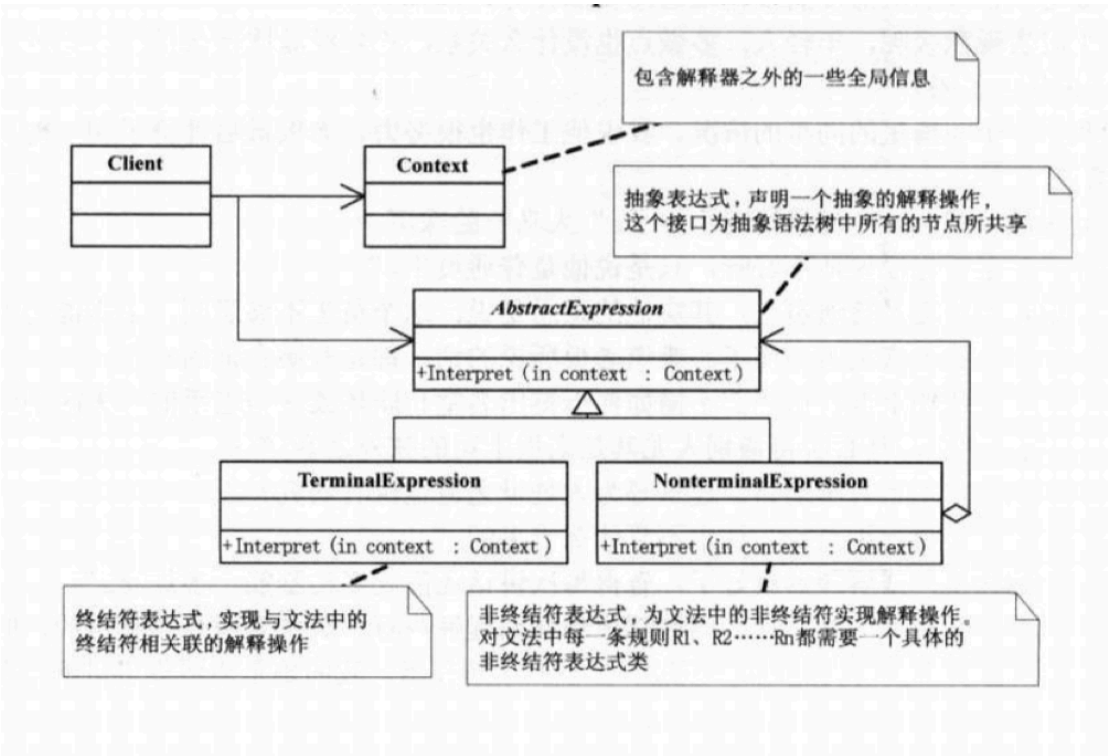
例如，对于如下表达式：

3+（（4*6）-（7/2））

输出：

3+（（4*6）-（7/2））=23.5

【实验原理】



解释器模式主要包含如下几个角色：

AbstractExpression：抽象表达式。声明一个抽象的解释操作，该接口为抽象语法树中所有的节点共享。

TerminalExpression：终结符表达式。实现与文法中的终结符相关的解释操作。实现抽象表达式中所要求的方法。文法中每一个终结符都有一个具体的终结表达式与之相对应。

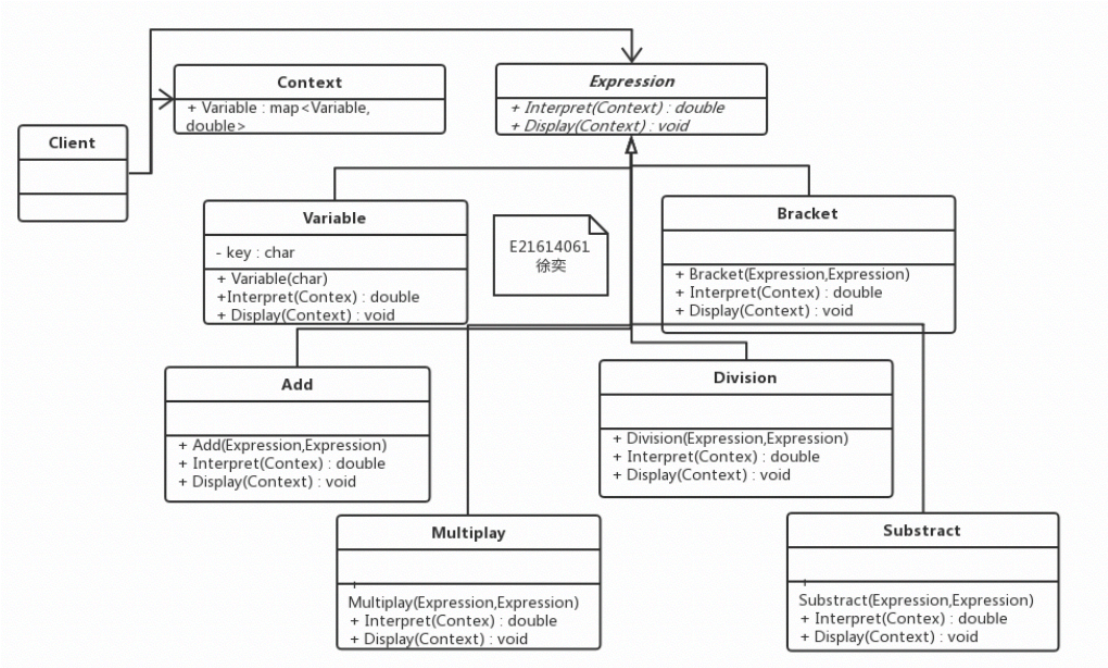
NonterminalExpression：非终结符表达式。为文法中的非终结符相关的解释操作。

Context：环境类。包含解释器之外的一些全局信息。

Client：客户类。

抽象语法树描述了如何构成一个复杂的句子，通过对抽象语法树的分析，可以识别出语言中的终结符和非终结符类。 在解释器模式中由于每一种终结符表达式、非终结符表达式都会有一个具体的实例与之相对应，所以系统的扩展性比较好。

【实验 UML 图】



【实验代码与函数】

为了使表达式显示出来需要增加一个括号类。

最高的抽象接口是 Expression,加减乘除括号变量常量均继承自 Expression 类。

```
#include<iostream>
#include<vector>
#include<map>
using namespace std;

class Context;
class Variable;

class Expression{
private:

public:
    virtual double interpret(Context *con) = 0;
    virtual void display(Context *con) = 0;
};

class Bracket :public Expression{
public:
    Bracket(Expression * n){
        exp = n;
    }
    double interpret(Context *con){
        return exp->interpret(con);
    }
    void display(Context *con){
        cout << "(";
        exp->display(con);
        cout << ")";
    }
private:
    Expression * exp;
};



class Context{
private:
```

```
    map<Variable*, double> valueMap;
public:
    void addValue(Variable *x, double y){
        double yi = y;
        valueMap[x]  = yi;
    }

    double LookupValue(Variable *x){
        double i = valueMap[x];
        return i;
    }
};

class Variable : public  Expression{
public:
    void display(Context *con){
        cout<<con->LookupValue(this);
    }
    double interpret(Context *con){
        return con->LookupValue(this);
    }
};


class Add : public Expression{
private:
    Expression *left, *right;
public:
    Add(Expression *leftx, Expression *rightx){
        left =  leftx;
        right = rightx;
    }
    void display(Context *con){
        left->display(con);
        cout<<" + ";
        right->display(con);
    }
    double interpret(Context *con){
        return left->interpret(con) + right->interpret(con);
    }
};

class Substract : public Expression{
private:
    Expression *left, *right;
```

```
public:
    Substract(Expression *leftx, Expression *rightx){
        left =  leftx;
        right = rightx;
    }
    void display(Context *con){
        left->display(con);
        cout<<" - ";
        right->display(con);
    }
    double interpret(Context *con){
        return left->interpret(con) - right->interpret(con);
    }
};

class Multiply : public Expression{
private:
    Expression *left, *right;
public:
    Multiply(Expression *leftx, Expression *rightx){
        left =  leftx;
        right = rightx;
    }
    void display(Context *con){
        left->display(con);
        cout<<" * ";
        right->display(con);
    }
    double interpret(Context *con){
        return left->interpret(con) * right->interpret(con);
    }
};

class Division : public Expression{
private:
    Expression *left, *right;
public:
    Division(Expression *leftx, Expression *rightx){
        left =  leftx;
        right = rightx;
    }
    void display(Context *con){
        left->display(con);
        cout<<" / ";
```

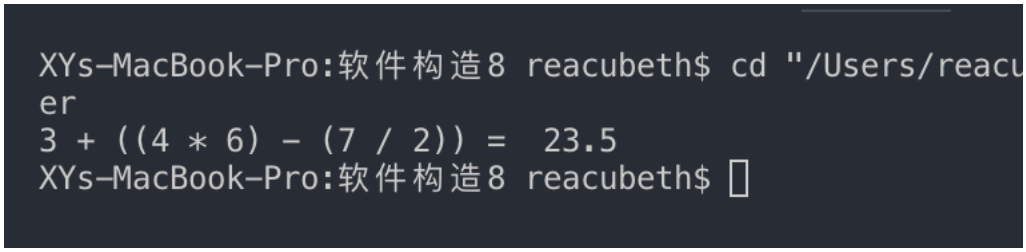
```
        right->display(con);
    }
    double interpret(Context *con){
        return left->interpret(con) / right->interpret(con);
    }
};



class Constant : public Expression{
private:
    int i;
public:
    Constant(int a){
        i = a;
    }
    double interpret(Context *con){
        return i;
    }
};


//3 + ((4*6) - (7/2))
int main(){
    Expression *ex;
    Context  *con  =  new Context();
    Variable *a = new Variable();
    Variable *b = new Variable();
    Variable *c = new Variable();
    Variable *d = new Variable();
    Variable *e = new Variable();
    con->addValue(a, 3);
    con->addValue(b, 4);
    con->addValue(c, 6);
    con->addValue(d, 7);
    con->addValue(e, 2);
    ex = new Add(a, new Bracket(new Substract(new Bracket(new
Multiply(b, c)), new Bracket(new Division(d, e)))));
    ex->display(con);
    cout<<" =  "<<ex->interpret(con)<<endl;
    return 0;
}
```

【实验结果】



【实验总结】

①本次实验掌握了解释器模式的基本实现。掌握了函数类的嵌套调用。

②Context 在本次实验中可以不用——除了上述用于表示表达式的类以外，通常在解释器模式中还提供了一个环境类 Context，用于存储一些全局信息，通常在 Context 中包含了一个 HashMap 或 ArrayList 等类型的集合对象（也可以直接由 HashMap 等集合类充当环境类），存储一系列公共信息，如变量名与值的映射关系（key/value）等，用于在进行具体的解释操作时从中获取相关信息。