

- 0 前言
- 1 第一次作业
  - 1.1 架构设计分析
  - 1.2 性能优化分析
  - 1.3 UML类图分析
  - 1.4 复杂度分析
  - 1.5 BUG分析
- 2 第二次作业
  - 2.1 架构设计分析
    - 2.1.1 多层括号嵌套
    - 2.1.2 自定义函数因子
    - 2.1.3 指数函数因子
  - 2.2 性能优化分析
  - 2.3 UML类图分析
  - 2.4 复杂度分析
  - 2.5 BUG分析
- 3 第三次作业
  - 3.1 架构设计分析
    - 3.1.1 用已定义函数定义新函数
    - 3.1.2 求导
  - 3.2 UML类图分析
  - 3.4 复杂度分析
  - 3.5 BUG分析
- 4 hack经验
- 5 感想
  - 5.1 心得体会
  - 5.2 反思
- 6 一些建议

---

## 0 前言

---

本单元的主题是表达式展开以及化简，目的是帮助我们培养层次化架构设计能力。在OO第一个单元中，我学习到了如何应用递归下降法，如何搭建评测机，如何编写更加面向对象的代码.....并且由于在最开始时选择了一个比较优秀的架构，所以在本单元作业中并没有进行重构。

在前言的剩余部分，我并不想讨论具体的作业完成方法或者什么所谓的java编程技巧，而是想从一个较高的视角来谈一下本次作业想让我们获得的能力——层次化架构设计能力。对于包括我在内的绝大多数普通人而言，当遇到一个复杂问题时是极其无力的。如果我们只是去驱使自己的身体去跑步，我们的大脑或许还能想明白在任意时刻我们的躯干处于何种状态，我们的四肢在做什么运动；但是如果是让我们去开飞机的话，我们即使穷尽脑汁也无法明确地知道任意时刻飞机的数以万计的零件处于何种状态，在做何种运动。但是事实上的确有人而且是很多人会开飞机的。这难道是因为他们记忆力超群吗，当然不是，而是因为飞行员根本没必要关注这些细节。

当我们面对这些复杂系统性工程时，我们就不得不对问题进行分层。还是以开飞机为例，飞行员在大多数情况下仅需要考虑飞机这个整体组成的对象，而无需考虑一个"微小"操作对某一个零件产生的"巨大"影响。这些零件的状态（属性）和动作（方法）实际上已经是更低层次的事情了，并非飞行员所应当考虑的。

我想如果掌握了层次化架构设计的能力，不仅能够较为顺利地完成本次作业，而且也对系统能力（学院一直想让我们获得的能力）的培养大有裨益。

# 1 第一次作业

## 1.1 架构设计分析

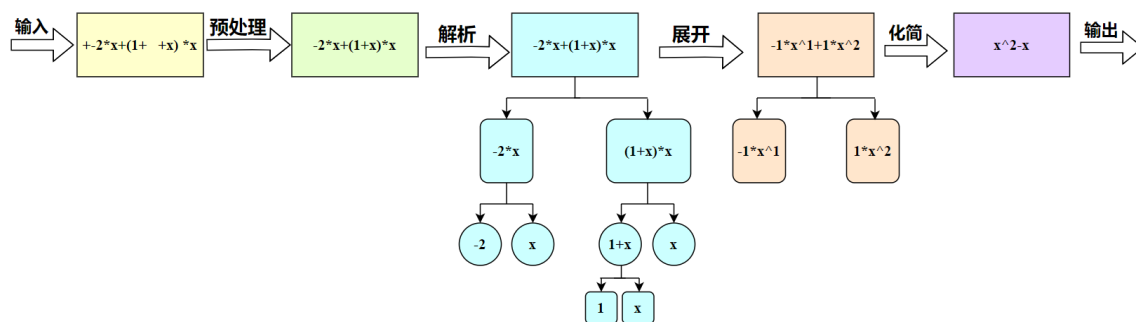
由于第二、三次作业的架构均是对本次架构的扩展，所以在此用较多篇幅介绍一下自己的架构设计思路。

本次作业是让我们展开一个最多嵌套一层括号的单变量多项式，多项式中项的因子只能是常数因子、幂函数因子、表达式因子，最后在正确展开的前提下再比较表达式的长度。

为了完成本次作业，我想我们至少需要考虑完成如下几个任务：

- 分析题干数据结构并**建类**
- **预处理**输入的字符串，得到方便后续操作的字符串
- **解析**预处理后的字符串
- **展开**有括号的表达式，得到一个多项式
- **优化**多项式并输出

为了能够让读者更好地理解这几个任务到底是什么，在这里给出一个样例：



对于**建类**任务，我们分析题目要求的输入与输出，会发现题目的输入很明显地存在**表达式**→**项**→**因子**的三层结构，而题目要求的输出很明显地存在**多项式 (Poly)**→**单项式 (Mono)**的二层结构。所以很自然地我们可以想到根据这种输入的限制与输出的要求建出相应的类。

对于**预处理**任务，我只做了两件事情：

- 删除输入字符串中的所有空白字符。
- 合并连续的符号。

对于**解析**任务，我本着拿来主义的原则选择了之前学长们大力推荐的**递归下降法**，也就是采用 `parseExpr()` → `parseTerm()` → `parseFactor()`，其中 `factor` 的种类有表达式因子、常数因子、幂函数因子。当解析到常数因子或者幂函数因子时则将其作为递归的终点。

对于**展开**任务，我首先在 `Expr`, `Term`, `ConFactor`, `PowFactor`, `ExprFactor` 中都实现了 `toPoly()` 方法，并在 `Poly` 类中实现了 `addPoly()`, `mulPoly()`, `mulPoly()` 方法。这样的话就可以将整个表达式递归转化成 `Poly`，而这个多项式是有许多形如  $a*x^b$  的 `Mono` 相加组成的。另外值得一提的是我在这里实现了对于 `Poly` 的深克隆的方法，虽然在第一次作业中没有用到，但是后续第二次作业、第三次作业都用上了。

```

public Poly deepClone() {
    ArrayList<Mono> newMonos = new ArrayList<>();
    for (Mono mono : this.monos) {
        newMonos.add(new Mono(mono.getCoef(), mono.getExp()));
    }
    return new Poly(newMonos);
}

```

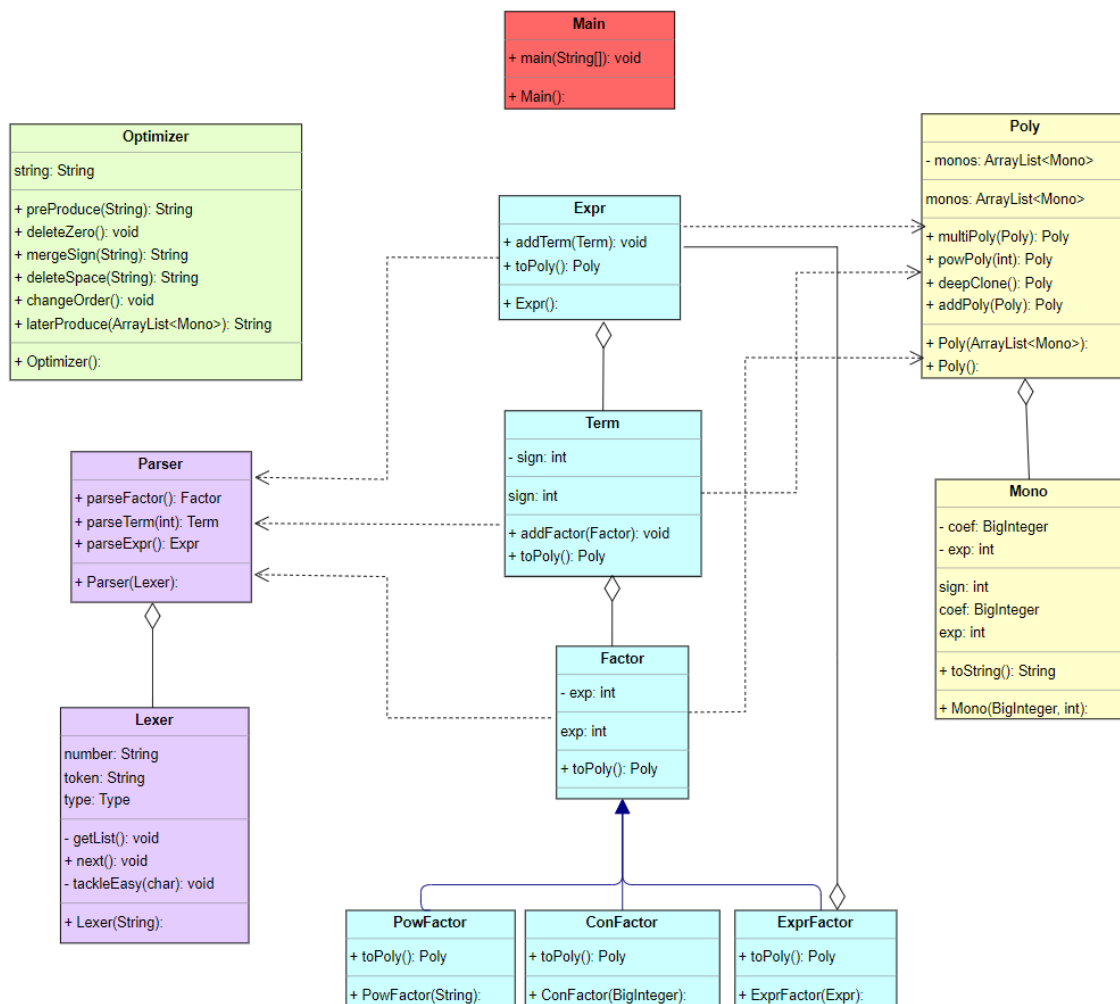
## 1.2 性能优化分析

如果上述建类、预处理、解析、展开过程中都没有bug的话，那么我们现在会得到一个符合如下格式的多项式  $\sum_{i=1}^n \text{coef}_i * x^{\text{exp}_i}$ ，并且其中的指数各不相同（因为在 `addPoly` 中就可以顺便实现**同类项合并**（在过程中合并同类项一举三得：减少长度+防止TLE+防止MLE））。

优化时我主要考虑了如下几点：

- 若  $\text{coef}_i$  为 1，则化为  $x^{\text{exp}_i}$
- 若  $\text{coef}_i$  为 -1，则化为  $-x^{\text{exp}_i}$
- 若  $\text{exp}_i$  为 0，则化为  $\text{coef}_i$
- 若  $\text{exp}_i$  为 1，则化为  $\text{coef}_i * x$
- 删除  $\text{coef}_i$  为 0 的项，当 `Poly` 中 `monos` 被删空时（即表达式本身是 0 的情况），向其中加入一个 0
- 由于会出现 `-x+1` 比 `1-x` 长的情况，因此这一步需要改变输出顺序使得第一项尽可能为正

## 1.3 UML类图分析



这些类大致可以分成五个部分，红色的是主类，是代码的运行入口；淡蓝色的是和输入相关的几个类；淡黄色的是和输出相关的几个类；淡紫色的是与解析相关的几个类；淡绿色的是对字符串或者表达式进行一些处理的类，包括预处理以及展开后的优化。

## 1.4 复杂度分析

Metrics Complexity metrics for 项目 'hw_1' from 周二, 19 3月 2024 ...				
Method metrics Class metrics Package metrics Module metrics Project metrics				
method	CogC	ev(G)	iv(G) ▾	v(G)
Ⓜ Lexer.getList()	22	1	9	15
Ⓜ Mono.toString()	14	9	9	9
Ⓜ Optimizer.mergeSign(String)	11	1	6	8
Ⓜ Lexer.tackleEasy(char)	5	1	6	6
Ⓜ Parser.parseExpr()	7	1	6	6
Ⓜ Poly.addPoly(Poly)	8	4	5	5
Ⓜ Optimizer.deleteZero()	4	1	4	4
Ⓜ Parser.parseFactor()	5	1	4	4
Ⓜ Lexer.getNumber()	2	1	3	3
Ⓜ Optimizer.changeOrder()	3	3	3	3
Ⓜ Poly.multiPoly(Poly)	3	1	3	3
Ⓜ Poly.powPoly(int)	2	2	2	3
Ⓜ Expr.toPoly()	1	1	2	2
Ⓜ Mono.getSign()	1	2	1	2
Ⓜ Optimizer.getString()	1	1	2	2
Ⓜ Parser.parseTerm(int)	1	1	2	2
Ⓜ Poly.deepClone()	1	1	2	2
Ⓜ Term.toPoly()	1	1	2	2
Ⓜ ConFactor.ConFactor(BigInteger)	0	1	1	1
Ⓜ ConFactor.toPoly()	0	1	1	1
Ⓜ Expr.Expr()	0	1	1	1
Ⓜ Expr.addTerm(Term)	0	1	1	1
Ⓜ ExprFactor.ExprFactor(Expr)	0	1	1	1
Ⓜ ExprFactor.toPoly()	0	1	1	1
Ⓜ Factor.Factor()	0	1	1	1
Ⓜ Factor.getExp()	0	1	1	1
Ⓜ Factor.setExp(int)	0	1	1	1
Ⓜ Lexer.Lexer(String)	0	1	1	1
Ⓜ Lexer.getToken()	0	1	1	1
Ⓜ Lexer.getType()	0	1	1	1
Ⓜ Lexer.next()	0	1	1	1
Ⓜ Main.main(String[])	0	1	1	1
Ⓜ Main.main(String[])	0	1	1	1

本次作业中复杂度很高的只有 `Lexer` 类中的 `getList()` 方法和 `Mono` 类中的 `toString()` 方法；我想第一个方法复杂度非常高的原因是其中含有许多个条件判断以将不同的词进行区分；第二个方法复杂度非常高的原因是因为我在其中实现了当系数或者指数为特殊值时的优化，所以会有许多分支，我想这是值得并应当允许的。

## 1.5 BUG分析

本次作业在中测、强测、互测中并未出现bug，在这里说一下给我身边的同学debug时记录下的一些bug：

- 我的一位朋友在展开或者说计算过程中并不会进行同类项的合并，那么会产生许多 `Mono` 或者 `Poly` 的对象，从而导致TLE和MLE的bug；
- 另一位朋友没有认真审题，在做完后才发现题目中幂函数的指数符号是 `^` 而不是 `**`。

## 2 第二次作业

### 2.1 架构设计分析

本次作业在难度上具有较大的跨度，概括来说主要有三个任务，第一个是使代码支持嵌套多层括号，第二个是使代码支持新增自定义函数因子，第三个是支持表达式中存在指数函数因子的情况；

#### 2.1.1 多层括号嵌套

对于第一个任务实际上由于使用了递归下降的方法来解析表达式，我们是不需要再做任何特殊处理即可完成这个任务。当然我在这里将指数的存储类型由int改为了BigInteger，因为多层嵌套括号的存在导致指数可能会超过int的范围的。

#### 2.1.2 自定义函数因子

首先在输入时将输入的函数定义式中的有效信息存储下来，至于什么是有效的信息呢？我想是指函数名和表达式之间的映射关系以及函数名和函数形参之间的映射关系。

然后我们接下来需要做的是当我们在解析表达式时遇到自定义函数时，将实参取代其中的形参后获得的一个字符串再解析成一个 Expr 即可，那么我们在这里实际上就实现了将表达式中的所有自定义函数都展开了，那么也就是实现了第二个任务。

```
else if (lexer.getType() == Lexer.Type.SELFFUN) {
    String funName = lexer.getToken();
    String string = Transform.replaceParas(funName, this.getArgs());
    factor = new SelfFunFactor(string);
}
```

在这个任务中，我新建了两个类(SelfFunFactor 与 Transform)，第一个是用于存储自定义函数这一因子，第二个是用于将自定义函数转化为常规的表达式。

#### 2.1.3 指数函数因子

在第二周周一晚上，我当时觉得这个任务会十分简单，似乎只需要给指数函数因子建一个 ExpFunFactor 类，然后再在 Mono 类中加上与指数函数相关的属性就能够完成第二个任务。

```
// ExprFactor.class
public class ExprFactor extends Factor {
    private Expr base; // 指数函数的指数表达式

    public ExprFactor(Expr base) {
        this.base = base;
    }
    // .....
}

// Mono.class
public class Mono {
    private BigInteger coef;
    private BigInteger powExp;
    private Poly exp; // 新增
    private BigInteger polyExp; // 新增

    // .....
}
```

```

    public boolean canAdd(Mono other) {
        if (!other.powExp.equals(this.powExp) ||
            !this.getExp().equals(other.getExp())) {
            return false;
        }
        return true;
    }

    // .....
}

```

但是后来我发现在加入指数函数因子后，如何合并同类项成为了一个问题。而这个问题的关键我想是如何判断两个指数函数因子内的指数部分是否相等。我之前对 `HashMap` 并不是特别熟悉，所以在第一次作业中使用了 `ArrayList` 存储多个 `Mono`，然后在之后又懒得改了一（技术债是这样的）。而 `ArrayList` 内的东西是有序的，但是实际上由于加法是满足交换律的所以这些 `Mono` 实际上是无序的，所以在判断是否相等时出现了麻烦。

但是我毕竟是一个懒惰的人，没有动力去修改之前的架构，所以最终决定采用递归的方式将两个指数部分相减并判断结果是否为0来判断是否相等。至于递归的终点在哪呢，我想应当是指数函数的指数为0的情况。

```

    public Poly subPoly(Poly elsePoly) {
        Poly ansPoly = this.deepClone();
        ArrayList<Mono> ansMonos = ansPoly.getMonos();
        ArrayList<Mono> elseMonos = elsePoly.getMonos();
        for (Mono elseMono : elseMonos) {
            int flag = 0;
            // 寻找是否有能够与elseMono相加减的项
            for (Mono ansMono : ansMonos) {
                if (ansMono.canAdd(elseMono)) { // 判断两个mono是否能够做加减运算
                    flag = 1;
                    ansMono.setCoef(elseMono.getCoef().subtract(ansMono.getCoef()));
                    break;
                }
            }
            // 如果没有能够与elseMono相加减的项，直接将elseMono取负后加入ansMonos
            // .....
        }
        return new Poly(Optimizer.deleteZero(ansMonos));
    }
}

```

我感觉这种合并同类项的方法的时间复杂度是极其高的，是不值得提倡的，不过所幸并没有在强测或者互测中出现tle的情况。

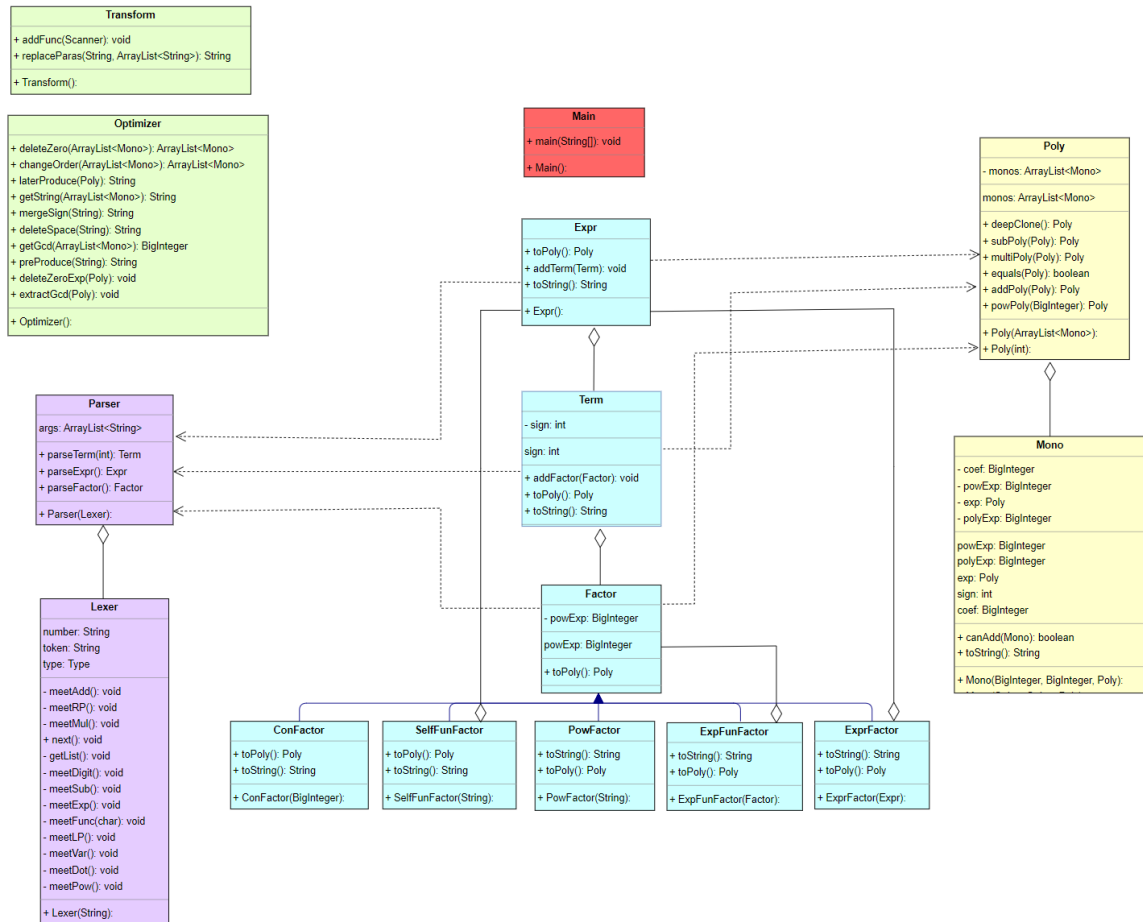
## 2.2 性能优化分析

在上次作业优化基础上，我主要是增加了对指数函数输出的优化操作：

- 当exp内为零时，则不输出该部分，比如  $2 \cdot x^2 \cdot \exp((0))$  转化成  $2 \cdot x^2$ ；
- 当exp内只有一个因子时，只输出一个括号，比如  $\exp((x^2))$  输出  $\exp(x^2)$ ；
- 提出exp内的最大公因数，如果提出后的长度小于提出前的长度，则输出提出最大公因数后的结果，否则则不改变，比如  $\exp((10000 \cdot x + 20000))$  将会转变成  $\exp((x+2))^{10000}$ 。



















## 2.3 UML类图分析

第二次作业的UML类图与上次在逻辑上基本一致，只是新增了 `Transform` 类、`SelfFunFactor` 以及 `ExpFunFactor` 类。



## 2.4 复杂度分析



Method metrics					
Method metrics		Class metrics	Package metrics	Module metrics	Project metrics
method		CogC	ev(G)	iv(G)	v(G)
 Mono.toString()		23	2	17	19
 Lexer.getList()		14	1	13	15
 Optimizer.getGcd(ArrayList<Mono>)		12	7	8	12
 Optimizer.mergeSign(String)		11	1	6	8
 Lexer.meetAdd()		4	1	7	7
 Poly.addPoly(Poly)		10	6	7	7
 Lexer.meetSub()		5	1	6	6
 Optimizer.deleteZeroExp(Poly)		6	4	5	6
 Optimizer.extractGcd(Poly)		11	2	5	6
 Parser.parseExpr()		7	1	6	6
 Parser.parseFactor()		8	1	6	6
 Poly.subPoly(Poly)		8	4	5	5
 Transform.addFunc(Scanner)		7	1	5	5
 Expr.toString()		5	1	4	4
 Optimizer.changeOrder(ArrayList<Mc		4	3	4	4
 Optimizer.deleteZero(ArrayList<Monc		4	1	4	4
 Lexer.getNumber()		2	1	3	3
 Mono.canAdd(Mono)		2	2	2	3

可以看到用于提公因数的 getGcd 方法的复杂度十分高，是因为在这个方法中不仅计算出了最大公因数是什么，还判断了提出后是否会变短。我感觉可以将这个方法拆分成两个相对独立的方法，一个用于计算最大公因数，一个根据最大公因数与指数部分判断是否提出最大公因数。

## 2.5 BUG分析

本次作业在中测、强测、互测中并未出现bug，在这里说一下互测房间同学以及身边朋友的bug：

- 有一个同学当输入0时程序会报错，原因是他在删除等于0的项后没有考虑所有项均为0的情况；

```
// input
0
0
// output
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: String
index out of range: 1
    at java.lang.String.charAt(String.java:658)
    at Operation.DeleteZero(Operation.java:42)
    at Main.main(Main.java:20)
```

- 有一个同学当输入一连串exp时会tle，原因是她在处理exp的幂指数时（即将  $\exp(\exp)^{\text{pow}}$  中的 pow 乘进去）时调用的是 powPoly() 方法而不是用 mulPoly() 方法直接将 pow 乘进去，这样会耗费十分多的时间。

```
// input
0
exp(exp(exp(exp(exp((x+1+exp(exp(x))))))))
// CPU_TIME_LIMIT_EXCEED
```

- 有一个朋友没有考虑到函数定义式的等号两边可以有空白字符，导致强测寄得很惨。（为他感到悲伤

## 3 第三次作业

### 3.1 架构设计分析

可能是因为我采用了一个较好的架构，我感觉第三次作业的难度远低于前两次，在周一晚上就完成了本次作业。第三次作业需要我们扩展两个任务，第一个是实现用已定义函数定义新函数，第二个是实现求导功能。

#### 3.1.1 用已定义函数定义新函数

我的第二次作业的代码实际上已经实现了这个功能，解释如下：

```
// 假如输入如下：
2
f(z)=z
g(y)=f(y)+y^2
g(x)
// 解析过程如下：
当解析到g时，会将x这个实参带入g的定义式，得到f(x)+x^2，然后再重新解析f(x)+x^2，然后解析到f时，
会将x这个实参带入f的定义式，得到x，从而将原表达式化为了x+x^2
```

#### 3.1.2 求导

对于求导的话，我首先新建了一个 `DerivativeFactor` 类，负责管理**求导因子**，在该类中用 `Expr base` 存出 `dx(.....)` 括号中的表达式。

做完前一步建类工作后，我在 `Expr`, `Term` 以及各种 `Factor` 的各个子类中都实现一个 `toDerivative()` 方法，且均返回 `Poly` 类型的对象。这个方法的作用是返回由该部分的导数而构成的 `Poly`。

I. 当  $f(x) = c$  ( $c$  为常数) 时,  $f'(x) = 0$

II. 当  $f(x) = x^n$  ( $n \neq 0$ ) 时,  $f'(x) = nx^{n-1}$

III. 当  $f(x) = \exp(x)$  时,  $f'(x) = \exp(x)$

V. 链式法则:  $[f(g(x))]' = f'(g(x))g'(x)$

VI. 乘法法则:  $[f(x)g(x)]' = f'(x)g(x) + f(x)g'(x)$

接下来我们就可以用课程组给我们的求导法则来写各个类的 `toDerivative` 了。

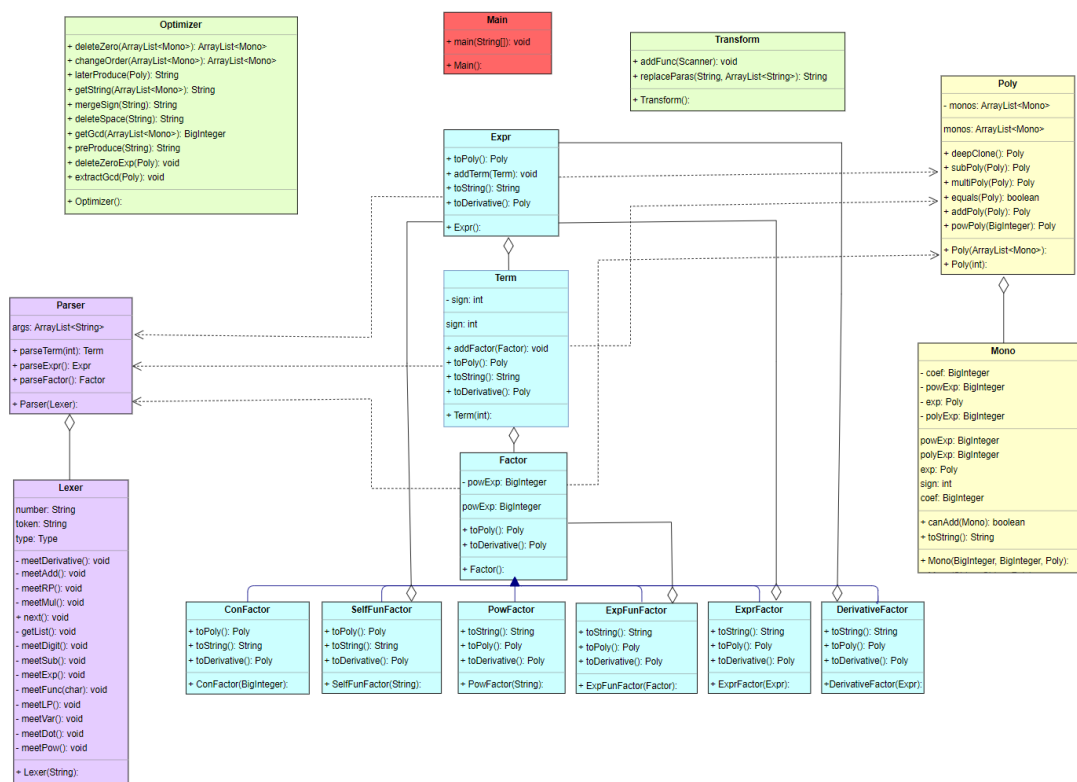
- 对于表达式，直接将各项用 `toDerivative()` 返回的 `Poly` 类型对象相加即可；
- 对于项，则应用导数的乘法法则即可；
- 对于常数因子，其求导后得到的值是0，则将0转化为 `Poly` 即可；
- 对于幂函数因子，应用图中方法2即可；
- 对于表达式因子和指数因子的处理比较复杂，但是也无非是链式法则和图中方法2的结合罢了。

这些处理方法再复杂也不过是对于上图公式的应用罢了，本身没有什么难度，值得一提的是如何求 `DerivativeFactor` 的导数，也就是遇到 `dx(DerivativeFactor)`，其中 `DerivativeFactor` 实际上是 `dx(Expr)`，这里实际上需要求 `Expr` 的二阶导，那么显然地我们应当先求内层的导数，然后再求外层的导数。

但是我们可以发现求内层导后会返回一个 `Poly` 类型对象，而我们求导的方法本身又无法处理 `Poly` 对象。所以我在这里先将获得的 `Poly` 对象转化成字符串（实际上就是将这个 `Poly` 当做最终的 `Poly`，走一遍优化和输出的流程即可，非常简便），然后再对这个字符串（实际上已经没有 `dx` 了）进行词法分析、语法分析，也就是重新走一遍之前的过程，得到一个新的 `Expr`，再对这个 `Expr` 求外层导数即可。虽然这样会使得求导所花费的时间变得十分之多，但是由于 `cost` 的限制，`dx` 无法使用许多次，所以也就采用了这种方法。

## 3.2 UML类图分析

第二次作业的UML类图与上次在逻辑上基本一致，只是新增了 `Transform` 类、`SelfFunFactor` 以及 `ExpFunFactor` 类。



## 3.4 复杂度分析

本次作业各个方法的复杂度和上次基本一致，值得一提的是 `Term` 类的求导方法的复杂度较高，原因是项的求导比较复杂，需要应用导数的乘法法则，也就需要嵌套两层 `for` 循环并搭配 `if-else` 判断语句。

method	✓	CogC	ev(G)	iv(G)	v(G)
<code>Term.toDerivative()</code>		7	1	4	4

```
public Poly toDerivative() {
    // 略
    // 外层每次循环对一个因子求导
    for (int iter1 = 0; iter1 < num; ++iter1) {
        // 略
        // 内存每次循环将不需求导的因子与需求导的因子相乘
        for (int iter2 = 0; iter2 < num; ++iter2) {
```

```

        // 需要求导
        if (iter1 == iter2) {
            temp = temp.multiPoly(factors.get(iter2).toDerivative());
        }
        // 不需要求导
        else {
            temp = temp.multiPoly(factors.get(iter2).toPoly());
        }
    }
    poly = poly.addPoly(temp);
}
return poly;
}

```

## 3.5 BUG分析

本次作业在中测、强测、互测中并未出现bug，在这里还是说一下互测房间同学的bug：

```

bug1: // 房间内一个人tle了
0
exp(exp(exp(exp(exp(x))))))

```

```

bug2: // 房间内一个输出了2，而应当输出1，是在处理导数时出现了小失误
0
dx(x*exp(0)^0)

```

## 4 hack经验

我认为hack应当有三种较为有效的方法，分别是用**评测机**自动测试、看**代码**找bug、**人工构造**一些极端数据用来测试。

对于自动评测机，一方面是要做到正确性判定，我认为[纪郅扬同学](#)的博客中提出的方法是足够科学有效的，所以也就使用了这种多人对拍的方法；另一方面是构建数据生成器，我想直接翻译题目的形式化表达即可，然后在睡觉时用自己的电脑或者服务器挂着对房间里的所有人进行评测（反正晚上有的是时间）。当然也可以通过设置常量池、调整与概率相关的参数来增大数据强度，以及控制嵌套层数等操作来防止出现长度或者cost过于大的数据。

```

def genDate(deepNum):
    string = getExpr(deepNum)
    if (len(string) > 1000):
        return genDate(deepNum)
    return string

def getExpr(deepNum):

def getTerm(deepNum):

def getFactor(deepNum):

def getConFactor():

def getPowFactor():

def getExprFactor(deepNum):

def getwhite():

def getZero():

```

对于看代码找bug这种行为，我想是十分值得的（如果有充足时间的话）。因为我们看其他人的代码不仅可以发现他们的bug，还可以学到他人优秀的架构以及一些java的知识。比如我之前并不非常清楚如何通过构建 ast 树来完成作业，后来在互测时看了他人的代码才了解到了这种做法。

对于最后一种方法，我感觉是最有用的（至少对于在互测中得分是最有用的）。因为6系许多人在进入强测前就已经用自己或者朋友的评测机测了上万次了，很难再通过评测机测出bug。所以我们应当构建一些比较特殊的数据来进行hack。那么什么是特殊的数据呢，我想有两种：

- 第一种是构建题目的文法与sympy文法不一致的数据：比如 $\exp((2*x))$ ，如果输出的是 $\exp(2*x)$ ，sympy也会认为这是对的，但这显然不符合我们题目的要求；
- 第二种是构建可能会让他人超时的极端数据，这就不得不提起李泰川同学构建出的数据了，竟然能够在如此苛刻的cost限制下构造出如此变态的测试数据，以至于我身边许多朋友都无法在时间要求内输出答案。

```

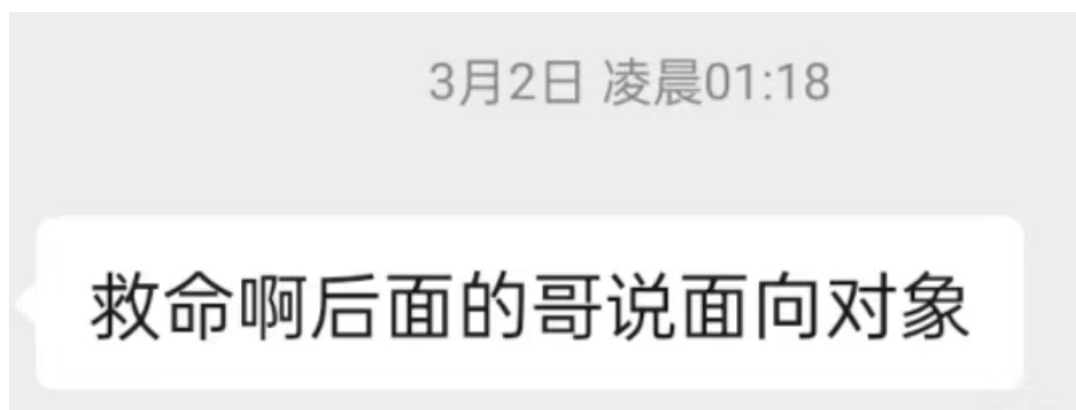
3
g(z)=exp(exp(exp(exp(z))))
f(y)=exp(exp(exp(exp(g(y)))))
h(x)=exp(exp(exp(exp(f(x)))))
h(f(g(exp(exp(exp(exp(x^8)))))))

```

## 5 感想

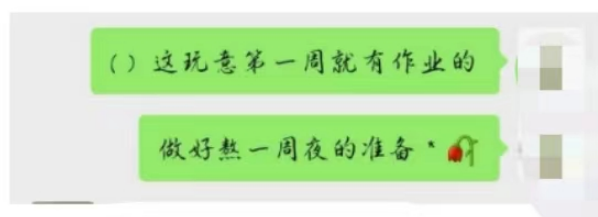
## 5.1 心得体会

首先第一单元真的非常非常累，上学期由于计组的压迫，我几乎不会记得当前是第几周，只会记得如今这一周是做第几个p的星期。这学期更甚，我甚至已经没了周几的概念，转而以今天是作业布置日、研讨课日、中测开放日、强测截止日、互测日亦或是bug修复日来分割我的一周。我身边亲近的人的生活似乎也都很累，被面向对象这门课占据着。



这个学期开学我走在路上 听见路过的同学  
都在谈论什么递归下降

？不是这不才开学第二天吗



不过这一单元确实让我学到了很多很多，下面是几点我从中学到的东西：

- 世俗地说，我感觉这门课是我考入北航至今以来与我的未来工作最相关的一门课，当然并不是说其他诸如计组、数分的课程不重要。只是感觉这门课对我的编程能力有着空前的提高。
- 然后，我感觉这门课程对我的系统思维能力的培养大有帮助，这也是我真正想在本科四年不断提高、不断追求的一种能力。我反而感觉诸如对于某门编程语言的掌握之类的能力并不是十分重要，可能我有时候就是一个这样不切实际的人。
- 还有一些可能不值一提的帮助就是这门课程让我对java、自动评测机等有了更进一步的了解。

## 5.2 反思

- 在本单元作业中我基本没用使用面向对象中的二十多种设计模式，我想之后的作业中我应当稍微刻意使用一些设计模式，以增强代码的可读性、可扩展性，并使自己熟悉各种设计模式。
- 写作业时经常发呆想其他事情，可能是因为最近确实有一些其他方面的事情，我想今后还是应当更专注一些。

## 6 一些建议

- 希望能够减少base分的占比，base分的存在让大多数提交数据的质量变得十分低。

- 感觉可以稍微放松一下互测时cost的限制，很多时候能够很明显地发现他人的bug，但是由于cost的限制使得每次都需要和cost斗智斗勇。
-