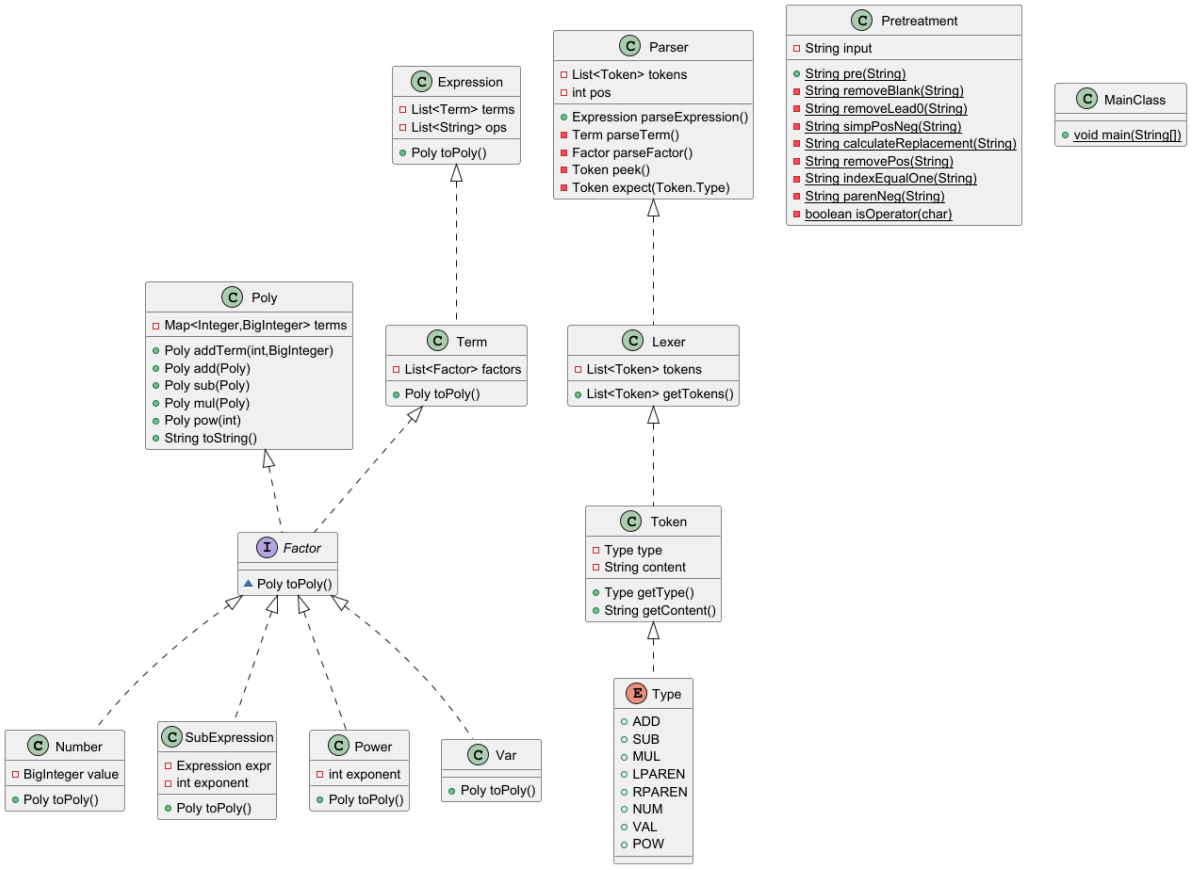
OO_Unit1总结

架构分析

第一次作业

第一次作业架构如下：



架构设计思路

主要沿用了OOpre中hw07的结构，采用相似的文法和词法解析方法，从Token到Lexer到Parser一脉相承，进行词法和文法解析。
在因子方面新增SubExpression类，用于处理表达式因子，不过这其实是多余的，可以直接将Expression类视为Factor接口的子类，因此仅第一次作业使用了SubExpression类，后续取消了该类。
为了便于处理多项式的合并和化简，设置多项式类Poly类，并为所有因子类、表达式类和项类增加toPoly()方法，使之能够转化为表达式，统一形式。考虑到最终表达式的形式是：
$\sum a * x^b$
在Poly类中使用HashMap储存单项式，键和值分别是指数和系数，便于化简，也因此我并没有单项式类Mono类。
设置预处理类Pretreatment类，用于实现以下功能：去除输入表达式中的空白项、化简连续的正负号、去除前导零、去除'^'和'*'后的正号、化简指数为1的项、处理'*''紧接'-'的情况。

程序的流程是预处理输入表达式 -> 解析词法 -> 解析文法 -> 转化为多项式 -> 合并化简 -> 输出

代码规模和复杂度

代码规模：

Source File	Total Lines	Source Code Lines	Source Code Lines [%]	Comment Lines	Comment Lines [%]	Blank Lines	Blank Lines [%]
Expression.java	24	21	88%	0	0%	3	12%
Factor.java	4	3	75%	0	0%	1	25%
Lexer.java	49	46	94%	0	0%	3	6%
MainClass.java	16	15	94%	0	0%	1	6%
Number.java	14	11	79%	0	0%	3	21%
Parser.java	108	98	91%	2	2%	8	7%
Poly.java	95	88	93%	0	0%	7	7%
Power.java	14	11	79%	0	0%	3	21%
Pretreatment.java	133	110	83%	8	6%	15	11%
SubExpression.java	15	13	87%	0	0%	2	13%
Term.java	18	15	83%	0	0%	3	17%
Token.java	21	17	81%	0	0%	4	19%
Var.java	8	7	88%	0	0%	1	12%

可以看出代码主要集中在预处理类、Poly类和文法解析Parser类中

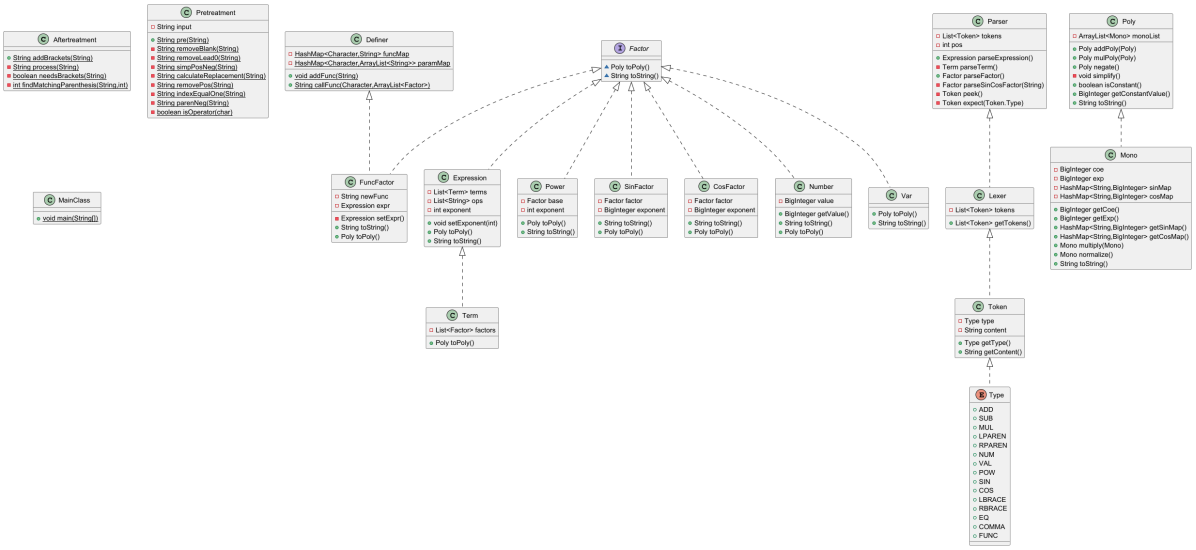
复杂度分析：

class	OCavg	OCmax	WMC
Expression	2.00	3	4
Lexer	6.00	11	12
MainClass	1.00	1	1
Number	1.00	1	2
Parser	3.50	9	21
Poly	3.83	11	23
Power	1.00	1	2
Pretreatme	2.30	4	23
SubExpress	1.00	1	2
Term	1.50	2	3
Token	1.00	1	3
Token.Type			0
Var	1.00	1	1
Total			**97**
Average	2.49	3.83	7.46

大多数类复杂度还是很好的，只有Lexer类、Parser类和Poly类复杂度偏高。Lexer类复杂度偏高是因为我在Lexer的一个方法中使用了大量的`if-else`语句来进行字符判断，其实可以将这个方法进行拆分，来降低复杂度；Parser类复杂度偏高是因为解析因子的方法parseFactor()中实现了对所有因子的解析，这其实也可以通过拆分这个方法为一些小方法来降低复杂度；Poly类复杂度偏高是因为Poly类中的toString()方法过长而且进行了大量的`if-else`判断，但是为了保证正确性最终也没有进行修改。

第二次作业

第二次作业架构如下：



架构设计思路

第二次作业由于增加了三角函数，最终表达式的形式变为了：
$\sum a * x^b * \prod sin(因子)^c * \prod cos(因子)^d$
第一次作业中使用HashMap来表示最小单元的方法已无法满足，因此索性进行了重构，主要的修改如下：
首先设置因子类SinFactor和CosFactor，储存三角函数，然后去除了SubExpression类，统一用Expression类来表示
设置单项式类Mono类，用来表示最小单元，Mono类中设置四个属性：BigInteger类型的系数和指数，以及用HashMap来储存的sin和cos，键是String类型的因子，值是指数。同时Poly类中不再用HashMap来储存最小单元，改用ArrayList来储存Mono，修改原有的化简合并逻辑。为Factor接口添加toString()方法，使得所有Factor子类能够转换为String储存在sin和cos的HashMap的键中.

```
\\ Mono.java
public class Mono {
    private BigInteger coe; // 系数
    private BigInteger exp; // 指数
    private HashMap<String, BigInteger> sinMap; //所有sin括号里的内容及其指数
    private HashMap<String, BigInteger> cosMap; //所有cos括号里的内容及其指数
    //其他方法
}
\\ Poly.java
public class Poly {
    private ArrayList<Mono> monoList;
    //其他用于化简合并表达式的方法
}
```

对于自定义递推函数，设置FuncFactor类作为Factor接口的子类，设置Definer类来进行字符串替换。处理自定义递推函数的策略是：在读入三行函数定义后调用Definer类的addFunc()方法，提取函数的形参列表和形参表达式，之后在解析表达式的时候遇到递推函数，对实参进行解析并加入到实参列表中，返回FuncFactor，在FuncFactor类中使用Definer类中的callFunc()方法，用实参替换对应的形参，最终返回解析完毕的表达式，处理完毕。

```
// FuncFactor.java
public class FuncFactor implements Factor {
```

```
    private String newFunc; // 将函数实参带入形参位置后的结果(字符串形式)
    private Expression expr; // 将 newFunc 解析成表达式后的结果

    public FuncFactor(String index, ArrayList<Factor> actualParas) {
        this.newFunc = Definer.callFunc(index.charAt(0), actualParas); // 获取形参
替换为实参后的表达式
        this.expr = setExpr(); // 对函数表达式进行解析
    }

    private Expression setExpr() {
        String s = Pretreatment.pre(newFunc); // 对字符串进行预处理
        Lexer lexer = new Lexer(s); // 词法解析
        Parser parser = new Parser(lexer.getTokens()); // 语法解析
        return parser.parseExpression();
    }
    // 其他方法
}
// Definer.java
public class Definer {
    private static HashMap<Character, String> funcMap = new HashMap<>(); // key存
储序号，value为定义式
    private static HashMap<Character, ArrayList<String>> paramMap = new HashMap<>
(); // key存储序号，value为形参列表

    public static void addFunc(String input) {
        //使用正则表达式提取函数定义式中的形参和形参表达式
    }

    public static String callFunc(Character index,ArrayList<Factor> actualParams)
{
        // 使用字符串替换递推获得f{2}~f{n}
    }
}
```

设置Aftertreatment类，对结果表达式进行后处理，对sin和cos括号内的表达式进行加括号。

代码规模和复杂度

代码规模：

Source File	Total Lines	Source Code Lines	Source Code Lines [%]	Comment Lines	Comment Lines [%]	Blank Lines	Blank Lines [%]
Aftertreatment.java	77	62	81%	3	4%	12	16%
CosFactor.java	43	38	88%	1	2%	4	9%
Definer.java	73	59	81%	3	4%	11	15%
Expression.java	46	40	87%	1	2%	5	11%
Factor.java	5	4	80%	0	0%	1	20%
FuncFactor.java	28	23	82%	0	0%	5	18%
Lexer.java	69	66	96%	0	0%	3	4%
MainClass.java	18	17	94%	0	0%	1	6%
Mono.java	105	90	86%	6	6%	9	9%
Number.java	25	20	80%	0	0%	5	20%
Parser.java	131	122	93%	1	1%	8	6%
Poly.java	111	98	88%	1	1%	12	11%
Power.java	28	24	86%	0	0%	4	14%
Pretreatment.java	133	110	83%	8	6%	15	11%
SinFactor.java	47	42	89%	1	2%	4	9%
Term.java	20	17	85%	0	0%	3	15%
Token.java	21	17	81%	0	0%	4	19%
Var.java	15	13	87%	0	0%	2	13%
Total:	**995**	**862**	**87%**	**25**	**3%**	**108**	**11%**

由于进行了重构而且新增了5个类，使得代码量增加了约400行，但是代码还是集中在预处理类、Poly类和Parser类以及Mono类中

复杂度分析：

class	OCavg	OCmax	WMC
Aftertreatm	4.00	5	16
CosFactor	2.33	4	7
Definer	4.00	5	8
Expression	1.75	4	7
FuncFactor	1.00	1	4
Lexer	9.50	18	19
MainClass	2.00	2	2
Mono	2.62	12	21
Number	1.00	1	4
Parser	3.86	12	27
Poly	2.56	7	23
Power	1.33	2	4
Pretreatme	2.30	4	23
SinFactor	2.67	5	8
Term	1.50	2	3
Token	1.00	1	3
Token.Type			0
Var	1.00	1	2
Total			**181**
Average	2.55	5.06	10.06

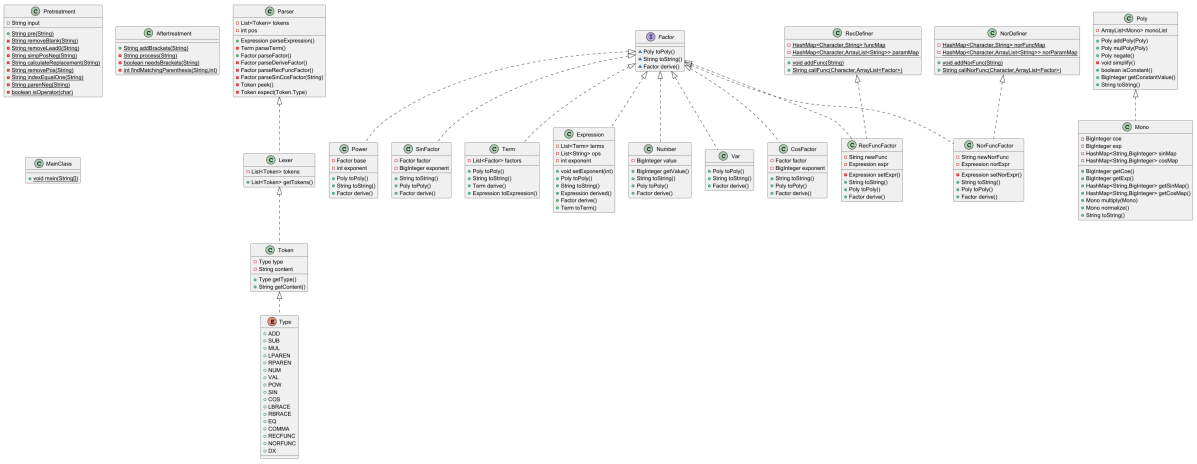
与第一次作业相比，由于设置了Mono类并且将Poly类中的部分功能并入Mono类，使得Poly类的复杂度降低不少。

但是由于依然采用大量的`if-else`来进行字符判断，使得Lexer类复杂度依旧非常高，由于第二次作业时间卡的特别紧，我并没有对其进行修改，直到第三次作业才进行优化。

Parser类复杂度高依旧是因为解析因子的方法parseFactor()过长，经过测试将其拆分为小方法可以有效降低复杂度。

第三次作业

第三次作业架构如下：



架构设计思路

本次作业较为简单，没有进行太多修改。
对于新增的自定义普通函数，其解决方式和第二次作业中的自定义递推函数相同，甚至要简单很多，因此我只是将第二次作业中的FuncFactor类和Definer类又复制了一份，改改名，简单修改了addFunc()方法和callFunc()方法就实现了这个要求。
对于求导因子，我并没有新建一个导数因子类，而是效仿第二次实验上机上的方法，在Factor接口中设置derive()求导方法，为每个因子类建立相应的求导方法便可，与第二次作业不同的是，我将Term类也归为因子，实现了万物皆因子。

代码规模和复杂度

代码规模：

Source File	Total Lines	Source Code Lines	Source Code Lines [%]	Comment Lines	Comment Lines [%]	Blank Lines	Blank Lines [%]
Aftertreatment.java	77	62	81%	3	4%	12	16%
CosFactor.java	64	57	89%	2	3%	5	8%
Expression.java	107	91	85%	7	7%	9	8%
Factor.java	7	5	71%	0	0%	2	29%
Lexer.java	53	46	87%	3	6%	4	8%
MainClass.java	23	22	96%	0	0%	1	4%
Mono.java	105	90	86%	6	6%	9	9%
NorDefiner.java	45	39	87%	2	4%	4	9%
NorFuncFactor.java	33	27	82%	0	0%	6	18%
Number.java	30	24	80%	0	0%	6	20%
Parser.java	160	149	93%	1	1%	10	6%
Poly.java	111	98	88%	1	1%	12	11%
Power.java	49	43	88%	1	2%	5	10%
Pretreatment.java	133	110	83%	8	6%	15	11%
RecDefiner.java	61	53	87%	3	5%	5	8%
RecFuncFactor.java	33	27	82%	0	0%	6	18%
SinFactor.java	67	60	90%	2	3%	5	7%
Term.java	57	50	88%	1	2%	6	11%
Token.java	22	18	82%	0	0%	4	18%
Var.java	20	17	85%	0	0%	3	15%
Total:	**1257**	**1088**	**87%**	**40**	**3%**	**129**	**10%**

与第二次作业相比增加了200行，并不多。
复杂度分析：

class	OCavg	OCmax	WMC
Aftertreatm	4.00	5	16
CosFactor	2.50	4	10
Expression	2.29	4	16
Lexer	4.50	8	9
MainClass	3.00	3	3
Mono	2.62	12	21
NorDefiner	3.00	3	6
NorFuncFa	1.00	1	5
Number	1.00	1	5
Parser	3.67	14	33
Poly	2.56	7	23
Power	2.00	4	8
Pretreatme	2.30	4	23
RecDefiner	4.50	6	9
RecFuncFa	1.00	1	5
SinFactor	2.75	5	11
Term	2.00	5	10
Token	1.00	1	3
Token.Type			0
Var	1.00	1	3
Total			**219**
Average	2.38	4.68	10.95

由于第二次作业中Lexer类复杂度过高，我对Lexer类中的核心方法进行了重写，不再采用直接粗暴的`if-else`判断，而是先判断字符的长度，再进行判断，这个方法有效的减小了Lexer类的复杂度。

其他类的复杂度和第二次作业相同。

Bug分析

第一次作业

第一次作业出现了一个非常不该出现的Bug，出现在预处理类当中。预处理类有个方法实现以下功能的方法：处理'*'紧接'-'的情况，解决方法是用括号将负项括起来，在判断负数后的符号时并未考虑右括号')'，导致出现如同下面样例的问题：
对于样例`(2*x*-1)^3`，经过我的预处理得到`(2*x*(-1)^3)`，显然是错误的，添加符号')'之后，这个Bug便解决了，这是个惨痛的教训。

第二次作业

第二次作业出现了两个Bug。一个是后处理类中加括号逻辑出现了错误，重写该类之后解决了这个问题；另一个Bug比较严重，Definer类的callFunc()方法用于实现用实参来替换递推表达式中的形参，由于实现时只是粗暴的进行字符串替换，导致出现以下问题：待展开表达式中的变量只能是x，如果形参顺序是先x后y，那会先进行含x实参替换x形参，再进行含x实参替换y形参，这不会出现问题；但如果形参顺序是先y后x，在用含x实参替换y形参之后，在进行用含x实参替换x形参时，会同时替换港灿替换完之后的y形参的部分，导致错误，解决方式是采用中间变量，无论第一次先替换哪个参量，都先替换为'z'，再进行第二次替换，在返回字符串的时候统一将'z'替换为'x'。

第三次作业

第三次作业出现了另外一个不该出现的Bug，在进行求导时，我没有考虑到指数为0的情况，只考虑到指数>=1的情况，导致只要对含有指数为0的表达式进行求导就会出错，为每个求导方法进行指数为0时的特判就解决了。

互测策略

第一次作业使用评测机生成一些复杂数据，同时自己构造特殊样例。后两次作业均靠自己构造一些复杂样例进行Hack。

优化策略

并没有进行太多的优化，只是在第二次作业中，实现了将sin(0)和cos(0)分别化简为0和1的优化。

心得体会

开学第一课有点太难了些😳，不过成就感还是有的，第一次一次性写这么多行代码。
我觉得这一单元的体会有如下几点：多和同学交流，无论是架构设计的经验或者Debug的经验还是一些好的样例；利用好上机实验，上机肯定不仅仅是填那些空白的地方，更多的是提供设计本次作业的思路；往届学长学姐能够提供很好的代码架构，可以多多学习；评测机确实很重要，能够找出自己意想不到的Bug，接下来要学习评测机的搭建，在第二单元可以尝试一下。
同时也感谢课程组在pre时提供递归下降的作业，没有pre时的接触第一次作业也会非常困难。

未来方向

1. 希望能够平衡一下作业难度，本单元第二次作业难度明显远高于另两次，难度应该向第三次作业偏离，因为第三次作业时是对架构理解最深的时候。
2. 在互测的时候输入的数据如果不合法，可以返回不合法的原因，如果是因为Cost过高导致，希望能够返回Cost，方便对数据进行调整。