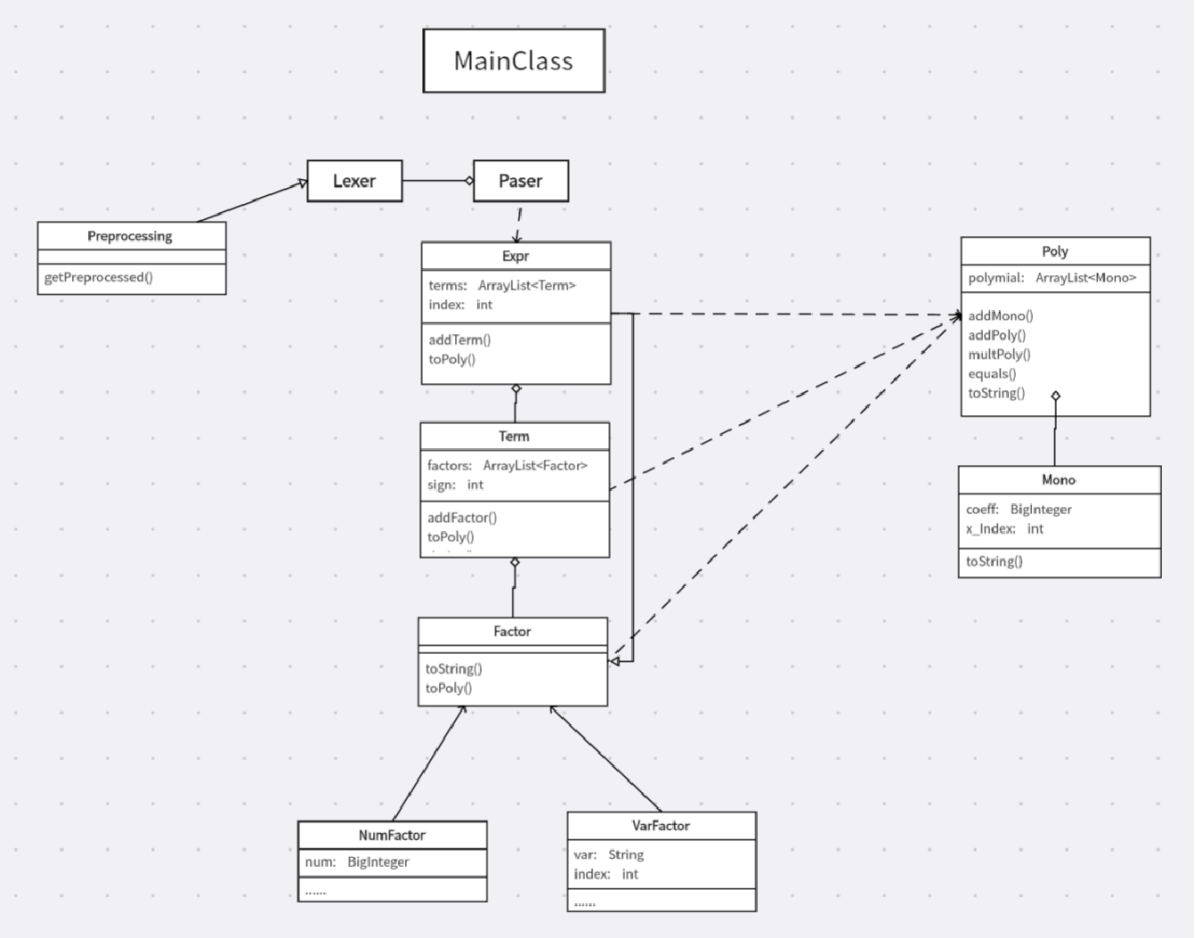
BUAA - OO - Unit1

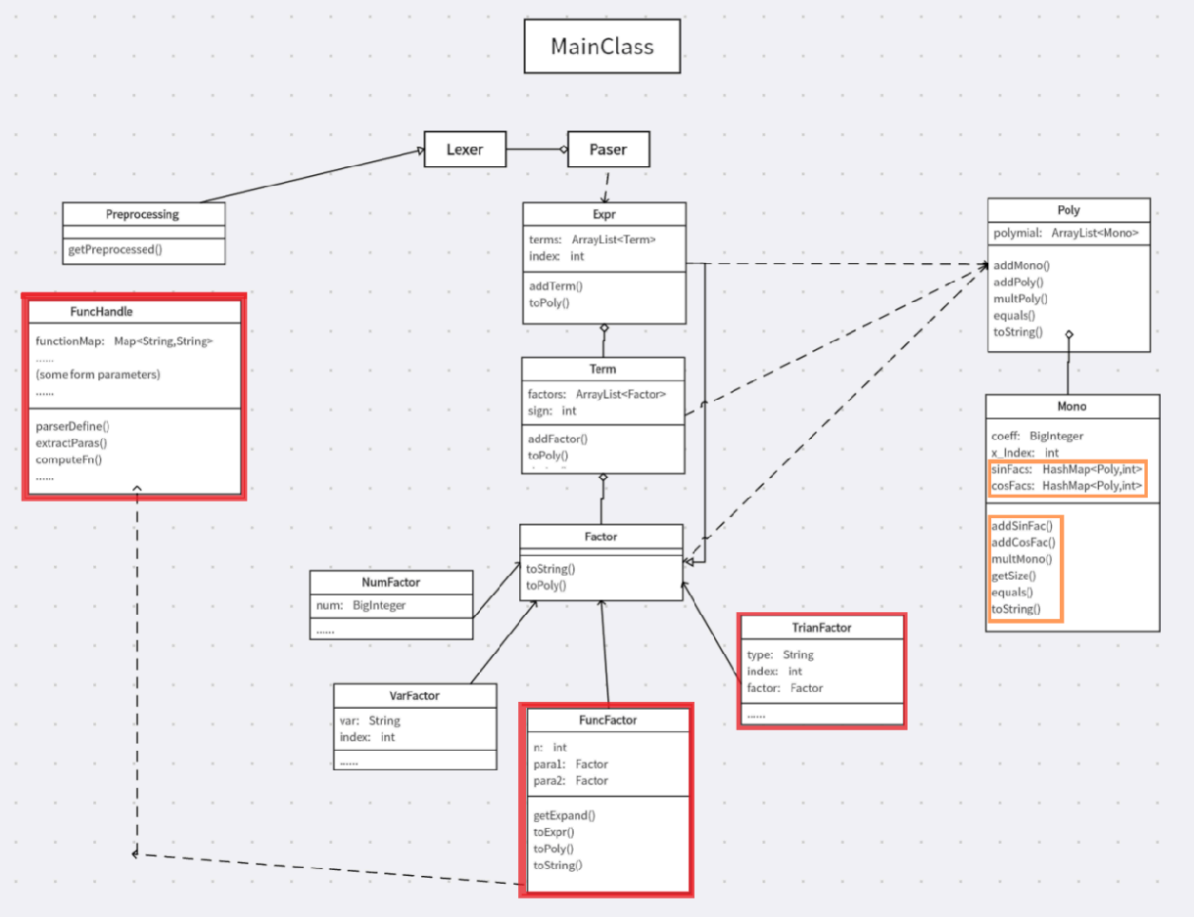
一、架构设计

1.1 第一次迭代



采用了递归向下的解析思路，解析完后转化成多项式Poly()，同时进行合并同类项的简化。最后输出化简后的多项式

1.2 第二次迭代



整体上新增了两个因子类 $TrianFactor$ 和 $FuncFactor$

- 由于 $TrianFactor$ 的加入，使得单项式 $Mono$ 不再像第一次作业那样形式简单统一，故对 $Mono$ 进行了修改，为其添加了两个HashMap类型的成员变量，使其能够统一于以下形式：

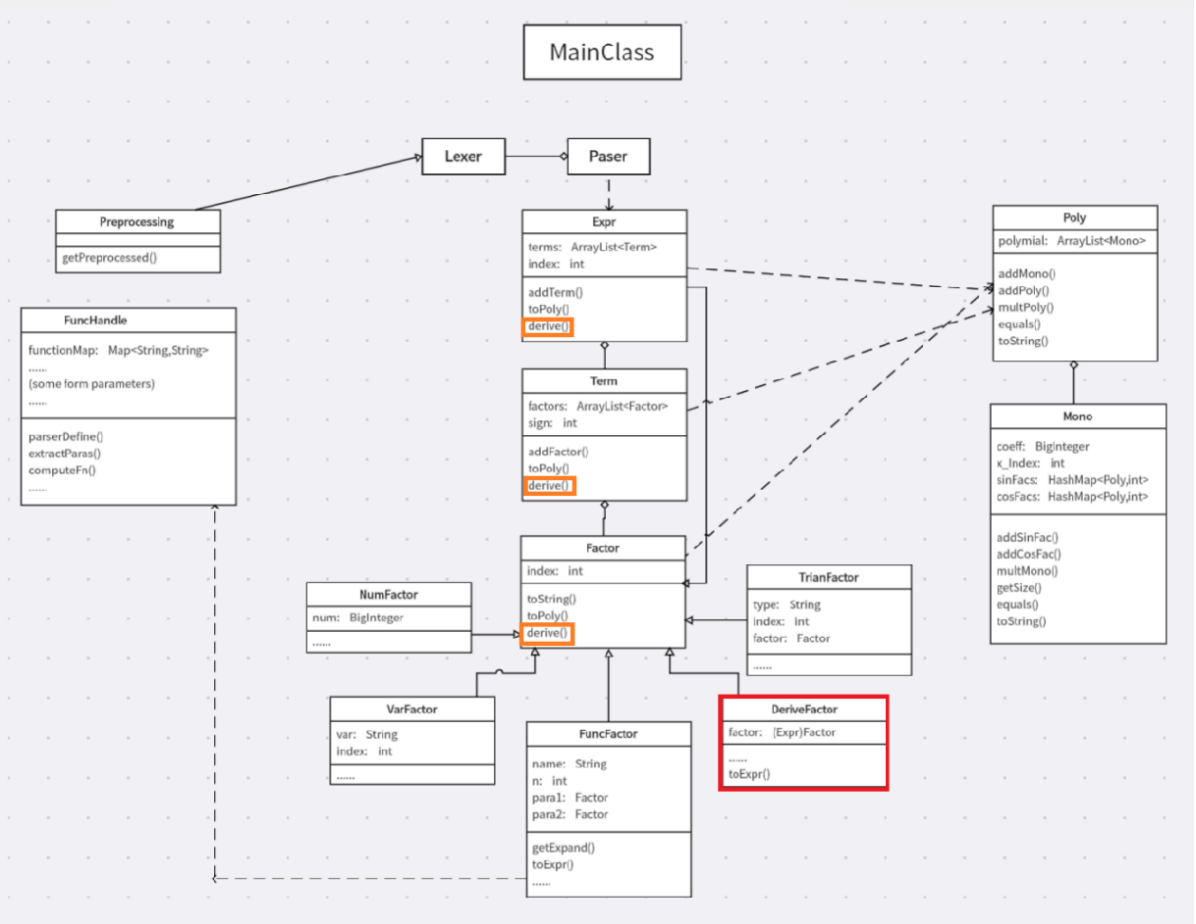
$$ax^n \prod_i sin(Factor_i) \prod_i cos(Factor_i)$$

  相应地，为了检查并合并同类项，需要增添一些方法，细节不再赘述

- 对于新增的递归函数，我写了工具类 $FuncHandle$ ，用来实现 获取定义、解析参数、替换参数、计算并储存函数表达式 等操作。工具类中的核心方法 `computeFn(int n , String arg1 , String arg2)`，功能是得到 `f{n}(arg1 , arg2)` 的展开式。思路是记忆化递归。

```
public static String computeFn(int n, String arg1, String arg2) {  0 个用法
    if (functionMap.containsKey(n)) {
        // 从 Map 中获取 f{n}(x,y) 表达式
        // x 替换为 arg1 , y 替换为 arg2
        // 返回替换结果
    } else {
        //根据定义式f{n}(x,y) = a*f{n-1}(.. , ..) + b*f{n-2}(.. , ..) + c
        //递归调用 computeFn(n-1 , .. , ..) 和 computeFn(n-2 , .. , ..)
        // 组合出 f{n}(x,y) 表达式，并存入 Map 中
        // x 替换为 arg1 , y 替换为 arg2
        // 返回结果
    }
}
```

1.3 第三次迭代



- 新增求导因子类 $DeriveFactor$ 。我选择 **将求导算子 `dx()` 视为一种因子**，对其的处理与对 $FuncFactor$ 的处理如出一辙，这使得我的结构非常清晰，写起来很舒适。然后我在 $Expr$、$Term$、各个 $Factor$ 类里新增了求导方法 `derive()`
- 另一个新增的自定义普通函数，对我的架构几乎没有任何影响，因为其地位与第二次迭代的递归函数`f{0}`和`f{1}`相同，仅格式上略有差异。

1.4 可扩展性

- 当新增其他种类的因子，如e指数、ln指数时，在最底层即他们自身的类里完成`toPoly()`、`toString`即可。而 $Poly$ 的基本计算单元是 $Mono$，只需为 $Mono$ 增添新的成员变量，使其适应统一形式，然后修改`equal`方法即可
- 当新增其他像函数、求导、简单积分等计算时，可以选择将其**视为一种因子**，与第二、三次迭代的处理方式相同。与函数类似的展开、字符串替换等需求，可交给工具类处理；与求导、积分类似的，从 $Expr$ 到 $Factor$ 自上而下地添加相关方法，向下调用即可

二、优化策略

2.1 结果长度优化

- 合并同类项：

  在`Mono`中重写`equal()`方法，逐个因子比较，由此判断是否为同类项

- `sin(-x) = - sin(x)`

  `cos(-x) = cos(x)`

2.2 运行时间优化

- 选择恰当的合并同类项的时机。`Poly`类的`addMono()`方法执行时，会向`poly`列表中添加`mono`，此时遍历`poly`列表，进行同类项的合并。我认为这是最佳的合并时机
- 对`Mono`类的`equal`方法做好剪枝。这无疑是时间复杂度最高的地方，因为要遍历两个Mono列表的每一个因子，而且要调用三角函数的“底数”`Poly`类型的`equal`方法，实际上是一个来回递归的过程，故剪枝变得十分有必要。简而言之就是选择适当顺序进行比较，若不同则立即返回`False`，避免接下来的冗杂比较。
- 对递推函数的展开采用上文所述的记忆化递归的方式，不再赘述

三、结构度量

使用了IEDA的插件 **MetricsReloaded** 进行分析，可以看到，`FuncHandle`、`Monomial`、`Polymial`的复杂度显著偏高

class	OCavg	OCmax	WMC
Expr	1.62	4	13
FuncFactor	1.38	2	11
FuncHandle	3.55	9	39
Lexer	2.17	5	13
MainClass	1.00	1	1
Monomial	3.92	15	51
Number	1.00	1	5
Parser	3.11	9	28
Polymial	3.75	9	30
Preprocessing	1.50	2	3
Term	2.00	4	12
Trian	1.86	5	13
Var	1.60	3	8
Total			**227**
Average	2.55	5.31	17.46

分析各个类的方法，找到导致复杂度偏高的原因

- `FuncHandle`类的`extractPara()`方法，模拟了栈，并结合正则表达式对定义式的参数进行提取，复杂度较高
- `Monomial`类和`Polymial`类的`equal()`方法，分支循环较多，导致复杂度偏高

四、Bug分析

本人在第二次作业中，提取递推表达式的参数时，先截掉了f{n-2}后面的函数表达式，然后进行了正则匹配，简单高效。然而在第三次作业中，我因为大意疏忽忘记了普通二元函数可出现在定义式的形参里，故没有修改方法。

尽管如此，强测依托助教的慈悲之心，我还是侥幸通过了所有数据点进入了A房，然后互测时被刀的体无完肤。

惨痛的互测结束后，我通过模拟栈来判断括号嵌套层数，进而解析函数的参数

五、Hack策略

为了便于对拍和互测，我用python写了评测机进行辅助

5.1 数据生成

采用递归向下的思路生成数据。为了便于 debug 和 适应互测要求，需要对数据进行限制，为此我设置了几个阈值控制数据范围

```
maxDepth = int(sys.argv[1])
maxExprLength = int(sys.argv[2])
maxTermLength = int(sys.argv[3])
maxIndex = int(sys.argv[4])
maxFuncDepth = int(sys.argv[5])
maxFuncnt = int(sys.argv[6])
```

```
def genExpr(depth,alpha):...

def genTerm(depth,alpha):...

def genFactor(depth,alpha):...
```

5.2 答案检查

我尝试过三种方式进行 $check$ ，个人认为第三种效果最佳

- **代入浮点数比较**

  与本次作业类似地写一个带入数值计算结果的项目，将输入和输出分别带入浮点数计算，选择合适精度比较是否相等

  此方案简单迅速，但经过实践，误差较大。精度在7位-16位之间都会出现 AC 和 WA 互相误判的情况，所以此方案最不可取

- **使用 python 里的 `sympy`。**

  此方案精度高，很少出错，而且几行便可写完。但是缺点也十分明显：运行速度太慢，使用UI界面后容易卡顿，故不算最佳

- **利用 `Poly` 的 `equal()` 方法与自己答案对拍。**

即修改第二次迭代的项目，删除函数相关内容（因为结果必不含 f ），然后将自己答案和待测答案解析成多项式后调用 `equal` 比较是否相同。

此方案效果最佳，兼具精度和效率。当然，需要保证自己代码的正确性，并且确保自己的化简程度更高（对于自动化随机评测来说，适度化简即可）

5.3 评测策略及外观

- 外观上我利用 `pygame` 库实现了UI界面，使得体验上更加舒适
- 首先是评测前进行阈值设定，控制生成数据的复杂度
- 设有普通模式和群测模式两种，前者用来1v1对拍，后者应用于互测
- 评测时可以采用随机生成数据和使用本地数据两种方式。本地数据主要是为了适应互测的Cost限制，对命中的随机数据作出修改再提交
- 使用多线程，同时评测多人多组数据，节约时间

六、心得体会

从写迭代作业，到写评测机debug，再到互测、修复，可以说，我的一周充满了OO。我投入了很多精力，也收获了很多心得

- 有一个**好的整体架构**便是成功了一半
- **慎用正则**，正则不是万能的
- 利用好**讨论区、研讨课、互测**，学习他人的优点
- debug时利用好**Junit单元测试、多和其他同学对拍、根据规则构造更全面的数据、极端的数据**

七、未来方向

可以考虑将自定义递推函数与自定义普通函数的迭代出现顺序互换，由简入难

其他都挺好的，感谢课程组老师和助教们的辛勤付出！