三.第二次作业分析

1.问题分析

本次作业相较上次增添了几个新功能，支持**自定义函数因子**和**三角函数因子**的解析计算，并支持**嵌套括号。**新增添文法规则大致如下；

> 变量因子 →→ 幂函数 | 三角函数 | 自定义函数调用 三角函数 →→ 'sin' '(' 因子 ')' [指数] | 'cos' '(' 因子 ')' [指数] 自定义函数定义 →→ 自定义函数名 '(' 自变量 [',' 自变量 [',' 自变量]] ')' '=' 函数表达式 例如 f(x)=x+1 自定义函数调用 →→ 自定义函数名 '(' 因子 [',' 因子 [',' 因子]] ')' 自定义函数名 →→ 'f' | 'g' | 'h' 函数表达式 →→ 表达式

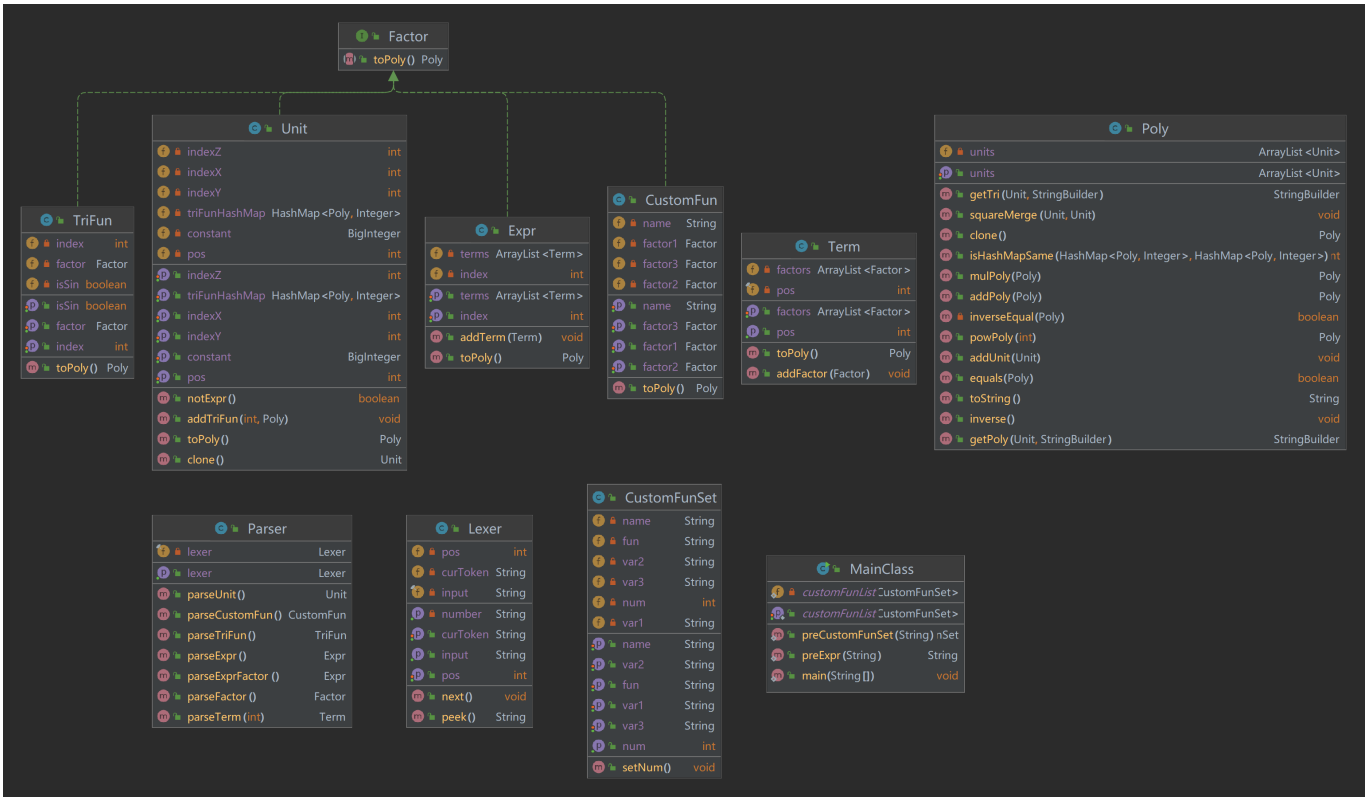
首先我们可以轻松解决嵌套括号的问题，按照上一次作业所示架构，已经可以支持嵌套括号的运算，可能使用正则表达式分析法的同学在面临这个问题时会显得更为麻烦?

接着我们来思考三角函数的实现，笔者认为有关三角函数的处理是本次作业(乃至本单元)最为核心且困难的实现。在第一次作业中，我们将每一个表达式，项和各种因子看作是多个基元集合组成的多项式，而每个基元仅用**5**个变量即可被完全描述。而本次新增了三角函数，以上概念明显不再可行，但就如何存储一个“基本表达式”而言仍能发现有相通之处。笔者将每一个基元新增一个属性，内容为HashMap<Poly, Integer>,**其中HashMap中每一个键值对代表着一个三角函数因子， 具体意义为Poly:存储的是该三角函数中含有的因子toPoly后的多项式Poly形式；Integer:存储的是该三角函数因子的指数，同时令正数为sin，负数为cos，这点很有创造性，使用一个HashMap即可存储任何形式的三角函数，而通过这一键值对即可完全描述出一个三角函数因子。**而Poly多项式类属性不变，仍然是ArrayList。拿$4*x**2*y*sin(x)*cos((4*z))$来说，前面单项式存储部分不变，后面跟着两个三角函数因子，使用HashMap对其指数和内部因子统一存储即可。以上讨论的是三角函数计算层面的实现，而解析表达式层面将在下文论述。

最后来考虑自定义函数的实现。我们可以新建一个CustomSet类来存储定义，具体存储其函数名，形参，函数表达式等属性。接着在我们解析表达式时会遇到自定义函数调用，笔者在此采用的是方法是**将其仍看作是一个因子解析进表达式树之中，而在自底向上toPoly时再将定义的函数与解析的函数相匹配，并进行形参到实参的带入**，这样做的好处是能在解析时专注解析，使目标表达式能按照文法完整的解析到表达式树之中，令人赏心悦目(也更简单)。

2.类分析

本次作业UML类图如图所示:



本次相较上次肉眼可见的复杂了许多(哭)，本次重点实现了CustomFunSet,CustomFun,TriFun类，并对Unit与Poly类进行了大的修改，其余类便不过多叙述。

(1)CustomFunSet 定义表达式类:

在本次作业中笔者学习到的一点**类要存储与之相关的所有信息，其属性必须能够全面的表示出一个类(对象)的状态。**本类主要是存储自定义函数的定义，具体属性上文有过描述，num属性的设置是为了能更快的得到该函数有几个形参，方便遍历。解析自定义函数定义的方法我运用了正则表达式(因为自定义函数定义格式较简单)，值得注意的是，自定义函数内容仍是一表达式，同样需要预处理，更甚者，同样可使用Lexer,Parser将其解析并再次转换为化简的字符串，如同题目要求对于表达式的处理与输出一样，这一点将在第三次作业的迭代中很为重要。

(2)CustomFun 自定义函数因子类:

本类实现Factor接口，可作为Factor统一存储。其中name，factor1~3存储着函数名与至多三个实参因子，并提供toPoly方法。

如何将自定义函数转换为Poly值得探索，我们要思考嵌套函数以及字符串替换有关的问题。笔者的方法是，先将其至多三个因子toPoly，若因子中仍是或内部仍存在自定义函数调用，则递归调用该函数即可(多态，递归的魅力)，这样嵌套函数的问题变迎刃而解。接下来处理实参至形参的转换，一个常见的错误便是直接replaceAll，如此做将导致已经转变成实参的变量被再次转变，因此笔者提供两种方法。第一种是字符串遍历法，由于不会对一个形参再次遍历，可以确保每个形参都仅被替换一次，第二种方法是可以在解析函数定义时将形参重命名为非xyz的变量。在形实参转换完毕后，便可将其看作一个完整的表达式，toPoly也是轻而易举。

(3)TriFun 三角函数类:

本类实现Factor接口，内部属性有isSin，标志该因子为sin or cos；指数index；以及三角函数内部的Factor。该类toPoly方法较为简单，新建Unit对象，并将内部Factor toPoly并与指数(注意正负)作为键值对一并加入空的

HashMap，再将Unit添加到新Poly中即可。

解析小结: 可以看到由于第一次作业的架构，即使新增更多因子，只需对Lexer进行小小的改动使其能识别词法，并对Parser中解析因子部分提供更多分支函数即可完成解析表达式的操作。其核心便在于基于BNF文法下的各个结构间的相互递归调用，使其能够在复杂的表达式组合上根据我们制定的**规则**有条不紊的解析，所谓大道至简，便是如此。

(5)Unit Poly:

首先要对toString方法进行扩展，遍历Array的同时，遍历Unit的HashMap输出三角函数的相关内容，而键值对里的Poly自然可通过递归调用Poly的toString方法得到结果~ Unit和Poly相互配合通过AddPoly，MulPoly，PowPoly等计算方法得到最终结果的思想仍不变，只不过本次要考虑每个Unit的HashMap，若不考虑对三角函数的化简，则对于带有HashMap的Unit的相加相乘也并非难事，本次作业到此为止即可得到功能的80分啦。

3.三角函数的化简

专门将本次作业的化简拎出来讲，是因为笔者大部分的时间都投入在了对于三角函数的化简上，为了最后20分的性能分(最后得到了15分+)，也算是值得。 本Part将分成两个部分介绍。

(1)基本化简 eg: sin(x) + sin(x) = 2 * sin(x)

(2)特殊化简

sin(x)2 + cos(x)**2 = 1**

sin((-x)) = -sin(x), cos((-x)) = cos(x)

sin(0) = 0, cos(0) = 1:

三角函数toPoly中对于Factor toPoly结果进行特判即可。

留给大家自主思考吧，有了基本化简的思路，实现以上

4.坑点及优化细节

(1)**不要过度优化 不要过度优化 不要过度优化** 过度优化将会给你带来意想不到的错误(来自一位朋友的惨烈教训)。**三角函数化简是无底洞**，一般来说对于三角函数的化简到上文为止即可，当然对于大佬们仍可考虑倍角公式，和差化积等高端操作。 (2)输出结果时 sin((x)) -> sin(x) sin((-x)) -> -sin(x) 即三角函数可以不加内层括号的便可以特判不加。 (3)注意自定义函数形参实参替换时左右加上"()"来保证运算顺序不变。 (4)若化简了sin(0),请注意sin(0)**0 的问题。 (5)本次作业是最**艰巨**的一关，克服他，第三次作业会柳暗花明。