Unit 1

一、前言

三周左右的学习，OO第一单元顺利结束了，个人认为有必要写个blog来反思总结一下自己第一单元的学习情况，以便更好地进行后面的学习。

~~之前从来没有写blog的习惯，这是自己第一次写blog~~~(^ - ^)

二、程序结构分析

CogC:它将一段代码被阅读和理解时的复杂程度，估算成一个具体数字.

- 出现"break"中止了线性的代码阅读理解，如出现循环、条件、try-catch、switch-case、一串的and or操作符、递归，以及jump to label：代码因此更复杂
- 多层嵌套结构：代码因此更复杂

ev(G):基本复杂度是用来衡量程序非结构化程度的，非结构成分降低了程序的质量，增加了代码的维护难度，使程序难于理解。

- 基本复杂度高意味着非结构化程度高，难以模块化和维护。
- 消除了一个错误有时会引起其他的错误

iv(G):模块设计复杂度是用来衡量模块判定结构，即模块和其他模块的调用关系。

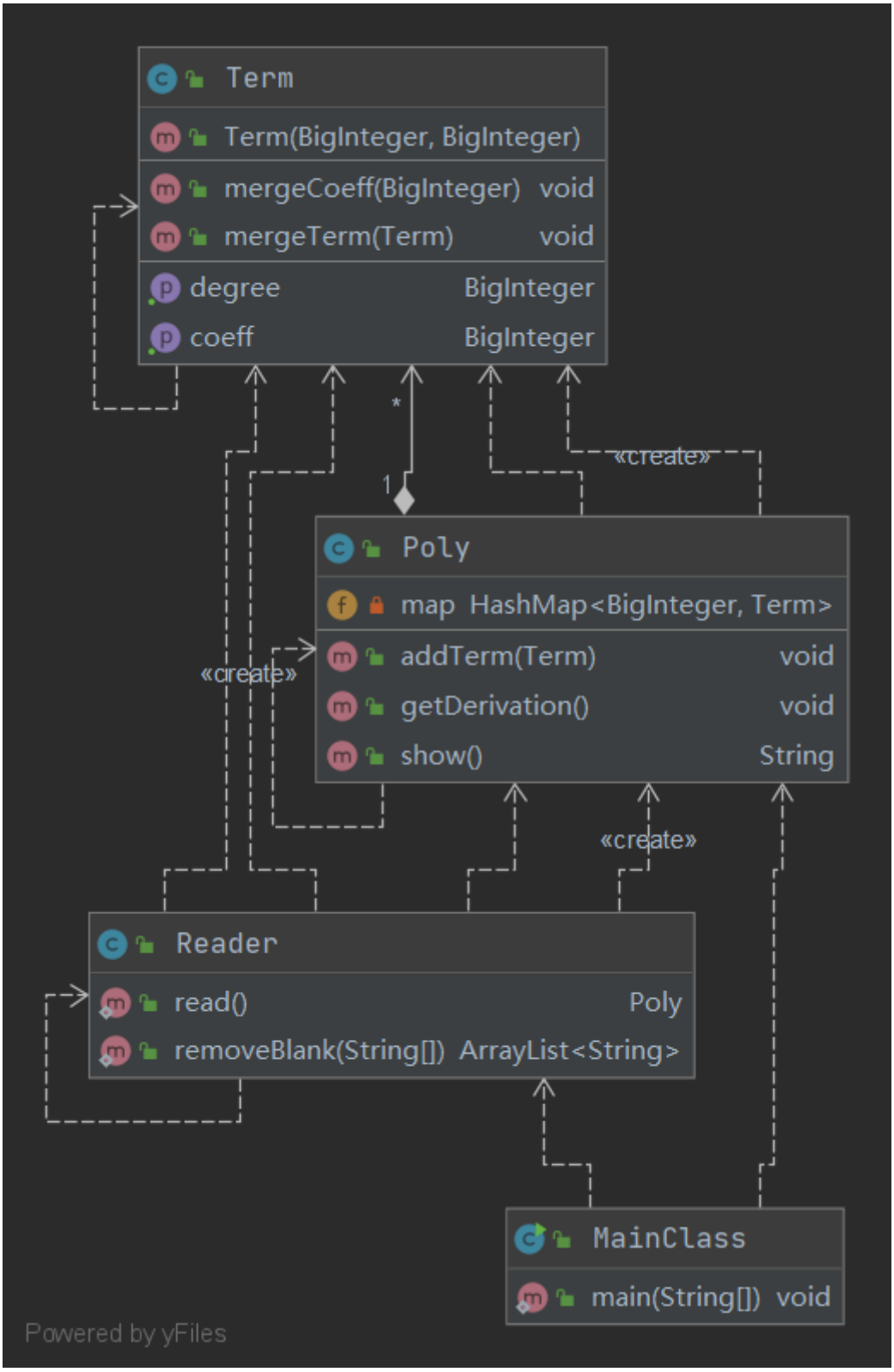
- 件模块设计复杂度高意味模块耦合度高，这将导致模块难于隔离、维护和复用。
- 模块设计复杂度是从模块流程图中移去那些不包含调用子模块的判定和循环结构后得出的圈复杂度，因此模块设计复杂度不能大于圈复杂度，通常是远小于圈复杂度。

v(G):是用来衡量一个模块判定结构的复杂程度，数量上表现为独立路径的条数，即合理的预防错误所需测试的最少路径条数

- 圈复杂度大说明程序代码可能质量低且难于测试和维护，经验表明，程序的可能错误和高的圈复杂度有着很大关系。

①.h1

1.UML



2.复杂度分析及设计意图

method	BRANCH ▲	CONTROL	CogC	ev(G)	iv(G)	v(G)
MainClass.main(String[])	0	0	0	1	1	1
Term.Term(BigInteger,BigInteger)	0	0	0	1	1	1
Term.getCoeff()	0	0	0	1	1	1
Term.getDegree()	0	0	0	1	1	1
Term.mergeCoeff(BigInteger)	0	0	0	1	1	1
Term.mergeTerm(Term)	0	0	0	1	1	1
Poly.addTerm(Term)	0	1	3	1	3	3
Reader.removeBlank(String[])	0	2	3	1	3	3
Poly.getDerivation()	0	3	8	1	4	4
Reader.read()	0	5	17	1	7	7
Poly.show()	3	12	26	6	10	11
Total	**3**	**23**	**57**	**16**	**33**	**34**
Average	0.27	2.09	5.18	1.45	3.00	3.09

MainClass

- **程序入口，控制程序的执行流程**（由于功能统一，后续不再介绍）
- 由于只设计到函数调用，复杂性很低

Reader

- **负责将读入字符串，经过处理生成Poly类之后进行返回**
- read方法利用正则表达式对字符串进行信息的提取
- removeBlank用于协助read进行字符串的信息提取
- 由于设计到正则表达式解析字符串，有各种匹配拆分操作，CogC较大

Poly

- **管理整个表达式，有一个HashMap作为属性，储存各个不同的Term类**
- addTerm，向Poly中添加新的Term，添加过程中会利用HashMap特点进行化简，用于构造Poly
- show打印Poly信息，有优化处理
- 复杂度良好

Term

- **有coeff，degree两个属性，最底层的基本存储形式**
- mergeTerm用于项的合并
- 复杂度良好

3.总体规模

Class	Attribute	Method	Lines
Reader	0	2	67
Poly	1	3	89
Term	2	5	27

4.优缺点

优点:

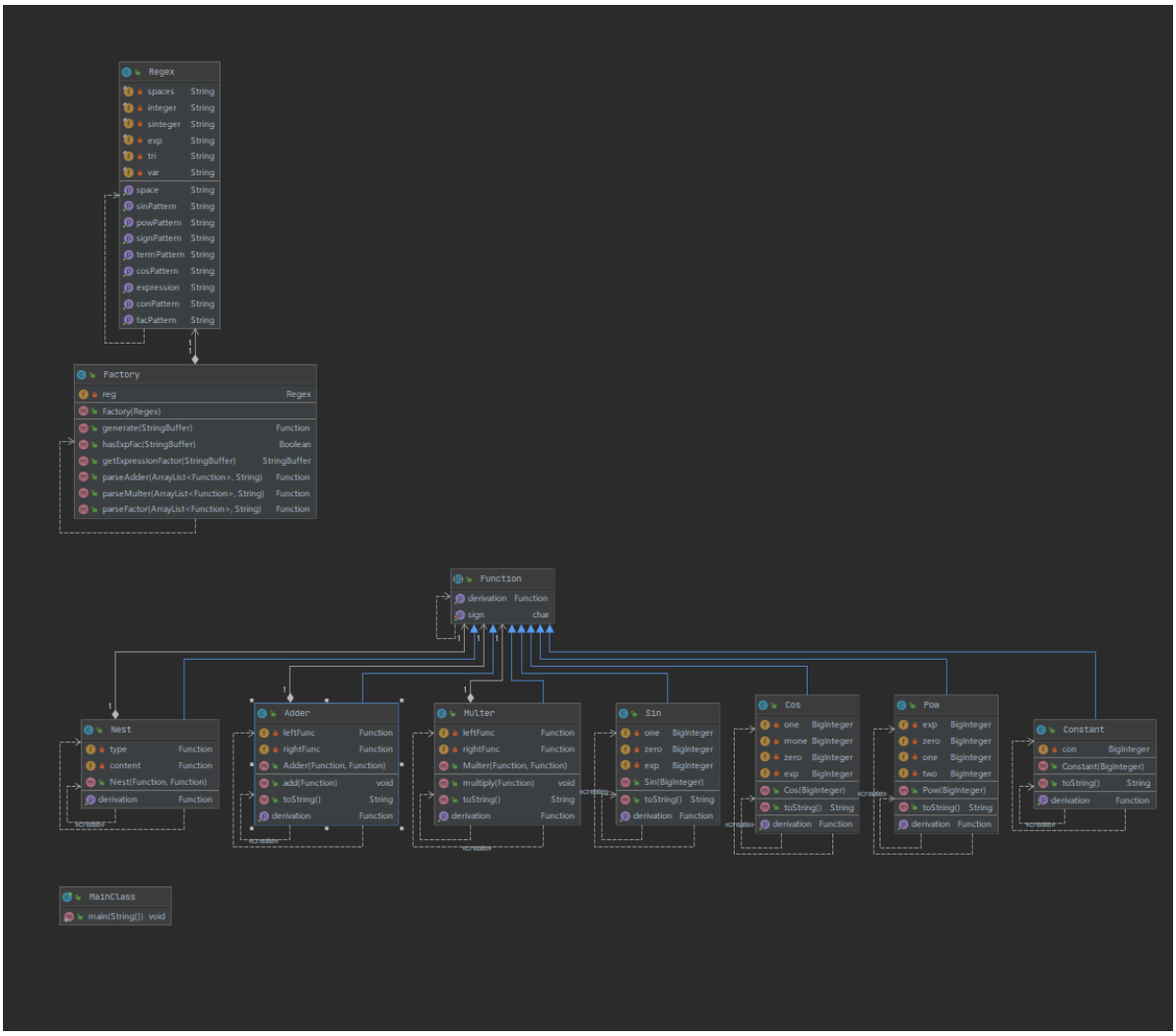
- 基本完成题目要求
- 能把读取信息的代码块从Main中提取出来单独成类，初步具有工厂模式的意识

缺点:

- 整体架构扩展性很差
- 在输出的时候不知道使用toString，而是直接进行了println

②.h2

1.UML



2.复杂度分析及设计意图

class	OCavg	OCmax	WMC
Adder	2.00	4	8
Constant	1.33	2	4
Cos	1.67	3	5
Factory	5.57	9	39
Function	1.50	2	3
MainClass	1.00	1	1
Multer	2.50	6	10
Nest	1.00	1	2
Pow	2.00	4	6
Regex	1.00	1	9
Sin	1.67	3	5
Total			**92**
Average	2.24	3.27	8.36

method	BRANCH	CONTROL	CogC	ev(G)	iv(G)	v(G
Adder.Adder(Function,Function)	0	0	0	1	1	1
Adder.add(Function)	0	0	0	1	1	1
Constant.Constant(BigInteger)	0	0	0	1	1	1
Constant.getDerivation()	0	0	0	1	1	1
Cos.Cos(BigInteger)	0	0	0	1	1	1
Cos.getDerivation()	0	0	0	1	1	1
Factory.Factory(Regex)	0	0	0	1	1	1
Function.getSign()	0	0	0	1	1	1
MainClass.main(String[])	0	0	0	1	1	1
Multer.Multer(Function,Function)	0	0	0	1	1	1
Multer.multiply(Function)	0	0	0	1	1	1
Nest.Nest(Function,Function)	0	0	0	1	1	1
Nest.getDerivation()	0	0	0	1	1	1
Pow.Pow(BigInteger)	0	0	0	1	1	1
Pow.getDerivation()	0	0	0	1	1	1
Regex.getConPattern()	0	0	0	1	1	1
Regex.getCosPattern()	0	0	0	1	1	1
Regex.getExpression()	0	0	0	1	1	1
Regex.getFacPattern()	0	0	0	1	1	1
Regex.getPowPattern()	0	0	0	1	1	1
Regex.getSignPattern()	0	0	0	1	1	1
Regex.getSinPattern()	0	0	0	1	1	1
Regex.getSpace()	0	0	0	1	1	1
Regex.getTermPattern()	0	0	0	1	1	1
Sin.Sin(BigInteger)	0	0	0	1	1	1
Sin.getDerivation()	0	0	0	1	1	1
Adder.getDerivation()	0	1	1	1	2	2
Constant.toString()	0	1	1	1	1	2
Factory.generate(StringBuffer)	0	1	2	1	2	2
Function.setSign(char)	0	1	2	1	1	2
Multer.getDerivation()	0	1	1	2	1	2
Cos.toString()	0	2	2	3	1	3
Sin.toString()	0	2	2	3	1	3
Adder.toString()	0	3	4	2	2	4
Pow.toString()	0	3	3	4	1	4
Factory.hasExpFac(StringBuffer)	0	5	12	6	5	7
Factory.parseMulter(ArrayList<Function>,Strinc	0	5	6	3	6	6
Multer.toString()	0	5	6	2	4	6
Factory.getExpressionFactor(StringBuffer)	1	7	18	4	4	12
Factory.parseFactor(ArrayList<Function>,String	0	7	14	8	8	8
Factory.parseAdder(ArrayList<Function>,String	0	8	11	2	10	10
Total	**1**	**52**	**85**	**69**	**75**	**99**
Average	0.02	1.27	2.07	1.68	1.83	2.41

Function

- **抽象类，各种函数都继承此抽象类，定义了公共的方法getDerivation，setSign用于项的符号处理**

Adder

- **继承Function，两个对象相加，对getDerivation进行了重写，以二叉树的形式存储**

Multer

- **继承Function，两个对象相乘，对getDerivation进行了重写，二叉树形式存储 与Adder相似**

Constant

- **继承Function，存储常数**

Pow

- **继承Function，存储幂函数**

Sin

- **继承Function，存储正弦函数**

Cos

- **继承Function，存储余弦函数与正弦函数类似**

Nest

- **继承Function，存储嵌套类，里面有两个属性，分别是外层函数，内层函数，为之后的嵌套扩展做准备**

Regex

- **由于采用正则解析字符串，涉及到各种模式串，统一用一个类进行管理**

Factory

- **简单工厂模式，对于传入的字符串返回一个Function对象，在解析的时候进行类似于递归下降的正则解析方法，对于头部的对象不断进行包装，完成二叉树建立**
- 同h1，由于采用正则匹配进行解析，导致复杂度一直较高

3.总体规模

Class	Attribute	Method	Lines
Regex	15	9	63
Factory	1	6	183
Function	1	2	17
Multer	2	4	48
Adder	2	4	40
Constant	1	3	22
Pow	4	3	32
Sin	3	3	30
Cos	4	3	31

4.优缺点

优点:

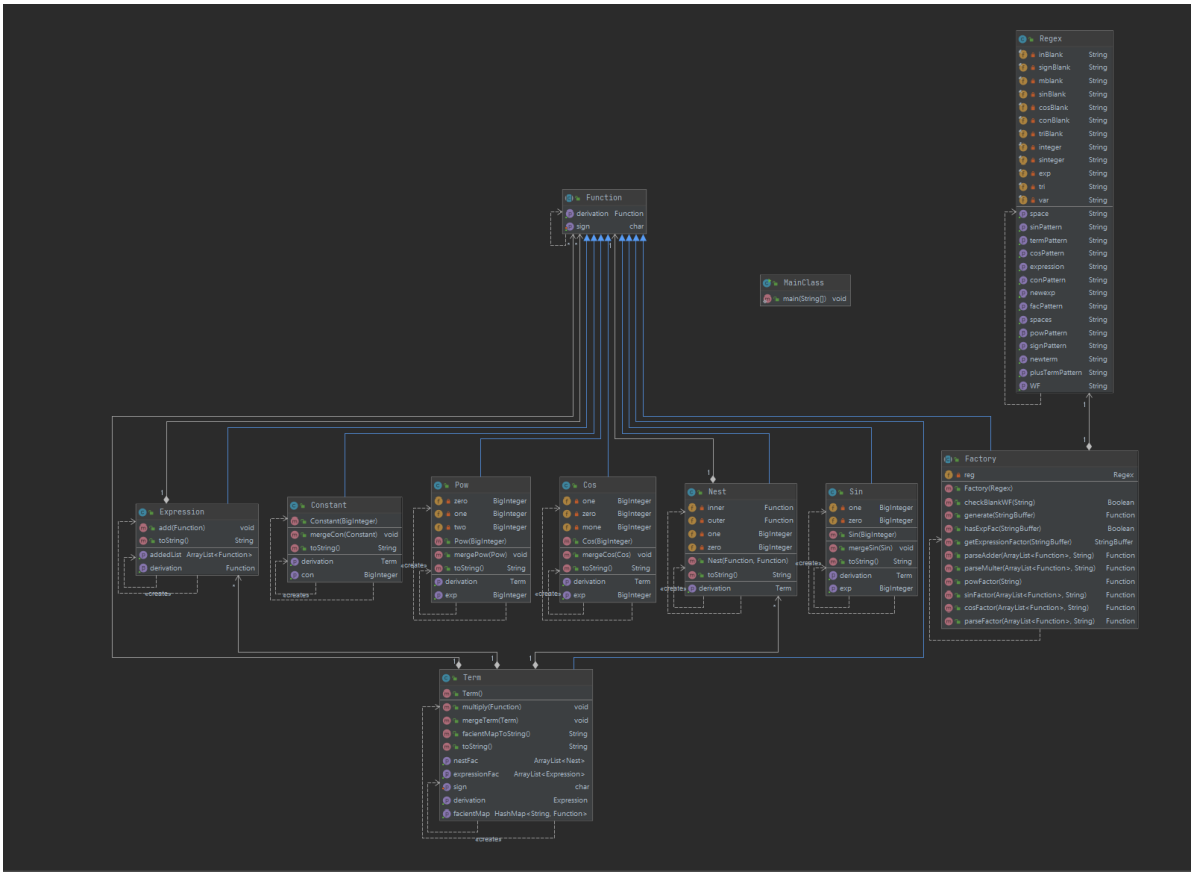
- 可以说这是我自认为最OO的一次了，基于二叉树设计数据结构，无论是建树过程还是getDerivation亦或是toString，都能很漂亮地通过递归完成，思路十分清晰，层次结构划分十分清楚，通过类的继承第一次体会到了真正意义上的OO

缺点:

- 由于采用了二叉树存储，在读入字符串形成对象的时候，需要对头部不断进行包装，导致的结果就是使以下这类本来十分简单的多项式被过度包装得十分复杂(层数增加非常快)
- $$x*x*x*x*x*x*x*x$$
- 同时二叉树存储导致化简极其困难，只要不属于同一个父节点就没办法化简（或许有办法，但我太菜没想到~）对于这样很容易合并的多项式也没法合并
- $$x+x**2-x**2$$

③. h3

1.UML



2.复杂度分析及设计意图

method	BRANCH ▲	CONTROL	CogC	ev(G)	iv(G)	v(G)
Constant.Constant(BigInteger)	0	0	0	1	1	1
Constant.getCon()	0	0	0	1	1	1
Constant.getDerivation()	0	0	0	1	1	1
Constant.mergeCon(Constant)	0	0	0	1	1	1
Constant.toString()	0	0	0	1	1	1
Cos.Cos(BigInteger)	0	0	0	1	1	1
Cos.getDerivation()	0	0	0	1	1	1
Cos.getExp()	0	0	0	1	1	1
Cos.mergeCos(Cos)	0	0	0	1	1	1
Expression.getAddedList()	0	0	0	1	1	1
Factory.Factory(Regex)	0	0	0	1	1	1
Function.getSign()	0	0	0	1	1	1
Nest.Nest(Function,Function)	0	0	0	1	1	1
Pow.Pow(BigInteger)	0	0	0	1	1	1
Pow.getDerivation()	0	0	0	1	1	1
Pow.getExp()	0	0	0	1	1	1
Pow.mergePow(Pow)	0	0	0	1	1	1
Regex.getConPattern()	0	0	0	1	1	1
Regex.getCosPattern()	0	0	0	1	1	1
Regex.getExpression()	0	0	0	1	1	1
Regex.getFacPattern()	0	0	0	1	1	1
Regex.getNewexp()	0	0	0	1	1	1
Regex.getNewterm()	0	0	0	1	1	1
Regex.getPlusTermPattern()	0	0	0	1	1	1
Regex.getPowPattern()	0	0	0	1	1	1
Regex.getSignPattern()	0	0	0	1	1	1
Regex.getSinPattern()	0	0	0	1	1	1
Regex.getSpace()	0	0	0	1	1	1
Regex.getSpaces()	0	0	0	1	1	1
Regex.getTermPattern()	0	0	0	1	1	1
Regex.getWF()	0	0	0	1	1	1
Sin.Sin(BigInteger)	0	0	0	1	1	1
Sin.getDerivation()	0	0	0	1	1	1
Sin.getExp()	0	0	0	1	1	1
Sin.mergeSin(Sin)	0	0	0	1	1	1

Term.Term()	0	0	0	1	1	1
Term.getExpressionFac()	0	0	0	1	1	1
Term.getFacientMap()	0	0	0	1	1	1
Term.getNestFac()	0	0	0	1	1	1
Factory.checkBlankWF(String)	0	1	1	2	1	2
Function.setSign(char)	0	1	2	1	1	2
MainClass.main(String[])	0	1	1	1	2	2
Nest.getDerivation()	0	1	2	1	2	2
Term.setSign(char)	0	1	1	1	2	2
Cos.toString()	0	2	2	3	1	3
Expression.add(Function)	0	2	4	1	3	3
Factory.generate(StringBuffer)	0	2	3	2	2	3
Factory.hasExpFac(StringBuffer)	0	2	3	3	2	3
Factory.powFactor(String)	0	2	4	3	2	3
Sin.toString()	0	2	2	3	1	3
Pow.toString()	0	3	3	4	1	4
Expression.getDerivation()	0	4	8	1	5	5
Factory.cosFactor(ArrayList<Function>,String)	0	4	10	5	2	5
Factory.sinFactor(ArrayList<Function>,String)	0	4	10	5	2	5
Term.mergeTerm(Term)	0	4	5	1	5	5
Factory.parseFactor(ArrayList<Function>,String	0	5	6	6	6	6
Expression.toString()	1	6	7	5	3	6
Factory.parseMulter(ArrayList<Function>,Strinç	0	6	7	3	6	7
Factory.getExpressionFactor(StringBuffer)	1	7	15	5	3	9
Factory.parseAdder(ArrayList<Function>,String	0	8	11	3	9	10
Term.facientMapToString()	0	8	17	1	6	9
Nest.toString()	0	9	30	8	9	11
Term.multiply(Function)	0	9	16	1	10	10
Term.toString()	0	11	19	6	9	12
Term.getDerivation()	4	20	34	11	15	17
Total	**6**	**125**	**223**	**125**	**149**	**188**
Average	0.09	1.92	3.43	1.92	2.29	2.89

---

h2到h3这里经过了一次重构，主要目的是解决之前**化简困难**的问题，本质上是地方没有什么变动，h2设计的架构已经具有了的扩展性

可以看出耦合度是最小的一次。

- 思路就是将之前的**二叉树**变成**多叉树**，二元运算变成连加、连乘，个人选择了ArrayList和HahsMap相结合的方式进行存储

Nest

- 继承Function，主要实现嵌套函数，由于嵌套只涉及到内层和外层（如果有多层还是可以看成内层和外层），所以这里可以选择二叉树结构，依据嵌套求导公式写好getDerivation就行了

Factory

- 先扫一遍检查是否有不合规范的空白符，除空白符之后，就可以递归地进行正则解析
- 遇到括号的时候递归调用parseExp将结果**入栈** 之后进行组装的时候**出栈** Factor扫描之后直接生成对象并组装成Term，Term出入栈组装成Exp
- 由于是通过形式化表述构造的模式串，所以不能匹配上则是格式错误，在parse的过程中随时抛出Exception处理WF

其他地方与h2完全相同

3.总体规模

Class	Attribute	Method	Lines
Function	1	3	17
Expression	1	4	66
Term	3	10	226
Constant	1	5	29
Pow	4	5	44
Sin	3	5	42
Cos	3	6	44
Factory	1	10	250

4.优缺点

优点

- 直接在h2基础上扩展，（~~虽然h2到h3重构了一次~~）但是大体框架还是没变~~

缺点

- Factory中没有采用正统的递归下降进行解析，而采用了地柜正则解析，导致为了正则的正确实现多了很多无谓的冗余操作，虽然道理简单，看似无懈可击，但实则操作困难，容易出bug。
- 为了能化简，我怎么感觉OO味淡了？感觉面向过程似乎更明显了一些?

三、个人bug分析

①h1

toString

在输出的时候产生部分错误，对于系数为0的项，自己断言已经从HashMap中remove了，在输出的时候没有给它带上符号

公测用例和互测用例中此Bug均有出现

原因:

- mergeTerm的时候处理有问题
- 输出的时候并不知道递归调用toString，导致逻辑极其混乱，容易产生bug
- 没有进行覆盖性测试

从复杂性分析可以看出来 Poly里的show方法圈复杂度相较其他方法已经高出很多了

②h2

toString

- **由于二叉树存储结构化简十分困难，思维混乱，在特定的情况会产生优化错误，丢失括号**
- toString调用之后没有进行存储，导致调用次数过多而超时

原因：

- 二叉树存储结构本身存在缺陷
- 对于递归调用的时间成本不敏感

总的来说并不是因为类或是方法复杂性过高导致的bug，而是由于二叉树存储这种结构先天的缺陷
~~主要还是我太菜~~

互测中也是以上两个bug被轮流hack.....

③h3

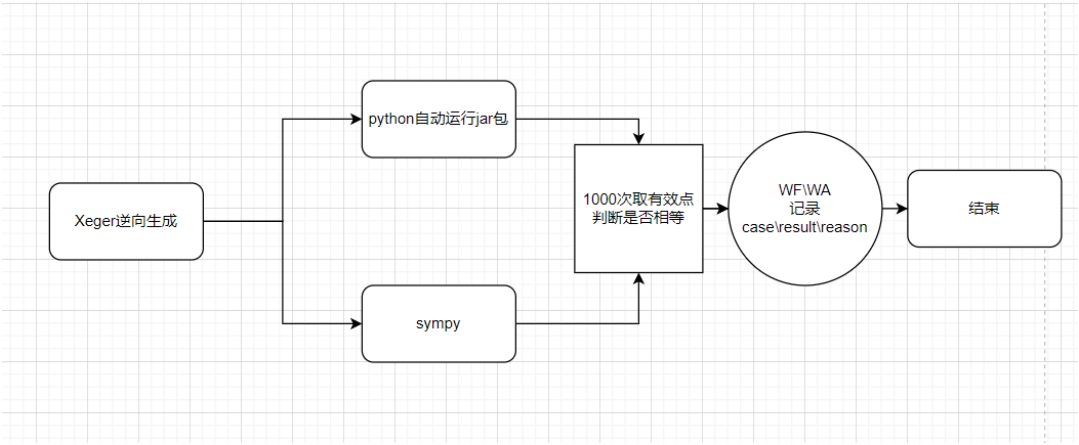
checkWF

- **WF在方法中有一个没判出来，**sin(+ 30)
  原因：
- 由于一来直接去了空白符，空白符没考虑完整导致后续模式串检查不能正确实现
  出现Bug主要是方法上的缺陷导致的，写了一个假递归下降，理解错了.....

由于互测不允许出现WF的数据，没被hack，逃过一劫

四、发现Bug的策略

- 1.**将自己在写作业时遇到过的坑记下来，等到互测的时候用来手动hack**
- 2.**写了python自动机进行随机测试**



随机生成数据，强度很低，我跑了2000轮才发现h2的bug.....
如果能够针对性地构造覆盖性的数据应该会好一点，但总比人工比对，手动黑盒测试效率高就是了
不过战果还是有的，前两次h中都能hack到较多的人，h3中也hack到许多，不过由于规则原因，都没法用就是了

3. **至于白盒测试，我仅仅在第一次中进行** 后面发现看完代码真的太费时间了~~并且还有许多很不好看的代码~~

由于本菜鸡不在高端局，**黑盒测试**效率远比**白盒测试**效率高

五、重构经历总结

一共经理了两次重构，分别是h1到h2，h2到h3

前面的部分已经列出了UML和复杂度分析~~就不再列啦ヾ(≧▽≦*)o

h1 -> h2

可以看出h1几乎没有什么扩展性，当时写h1的时候其实就已经做好了重构的准备了，h2中加入了层次划分，继承等OO的方法进行框架构造，可以说是思路最清晰的一次，OO味最浓的一次，可以采用了二叉树存储，导致化简很困难

h2 -> h3

由于h2扩展性较好，h3在架构层面几乎不需要更改，只是需要改一改数据结构，并对数据结构做一下方法在接口上的调整，使得能顺利化简，当然只是只最基本的化简，三角函数那部分我没有去考虑.....

六、心得体会

- 从面向过程的思想转变为面向对象的思想对于我这种面相过程都不太行的人来说是较为困难的，所以其实有很多地方都是只是披着OO的皮.....
- 但是不可否认的是，有些工作，不用面向对象的思路来完成是十分困难的，~~比如第一单元作业~~
- 最好在做第一次作业的时候就要注意扩展性，否则后面又得花大量时间重构
- 选对方法很重要，~~不要一头撸死~~，用了一个假递归下降头铁到底，实际上真正去请教大佬递归下降可能并不会花那么多时间研究正则，不断调试，虽然正则能力得到了极大提升 (￣﹀￣)↗
- Java能力得到了较大的提升
- 多读同学的优秀代码，学习优秀同学的做法~~虽然正则头铁到底这种思路是h1某位同学那受到启发的~~
- 如果开始没有做好扩展性，只能重构，那么不要吝啬，适当重构不仅能提升code能力，同时还能让自己对于架构的理解更加清晰