# 实验目的

通过⾃⼰编写、调试⼀个词法分析程序，并对语句进⾏词法分析，进行更好理解的词法分析原理。

# 内容描述

本程序目的是进行一个对java语言程序的词法识别，可识别保留字、变量名、操作符、数字等等，并输出格式为 （id ， content） 的 TOKEN序列，同时对未定义字符、整型过大、变量名格式错误等异常进⾏异常报错。

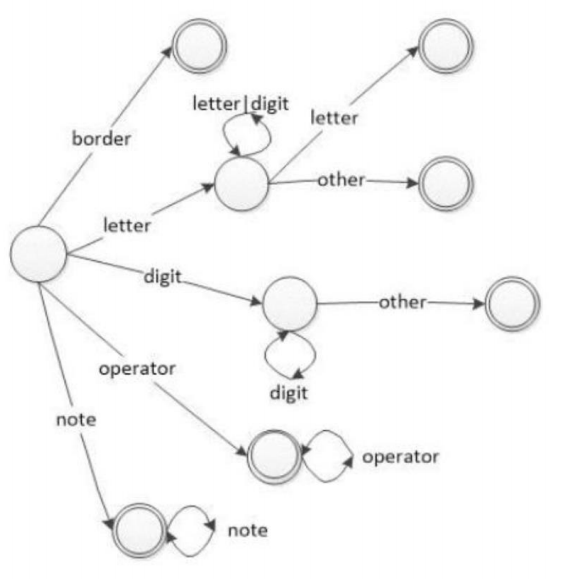
# 思路方法

1. 针对要识别的单词符号写出正则表达式
2. 构造出正则表达式对应的NFA
3. 合并所有 NFA 并化简为 DFA
4. 基于DFA编写代码
5. 代码中具体的实现：先读取⼀个输⼊字符，判断其可能的类别，再读取下⼀个继续判断，若已经识别出则添加到输出链，指针指向下一个字符位置，否则继续依次读取下⼀个字符。

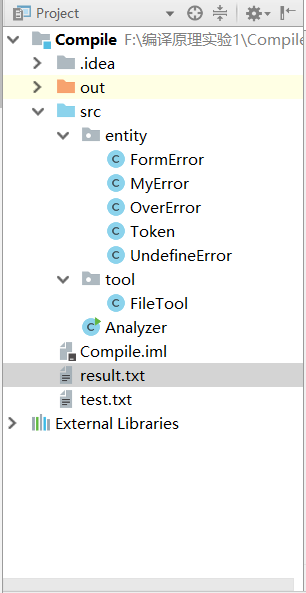
# 假设

假设输出的文件内容是正常的java程序，即包含合法的保留字和运算符。

# 相关FA



# 重要数据结构描述



总共有7个类。其中MyError是FormError,OverError和UndefineError的父类。

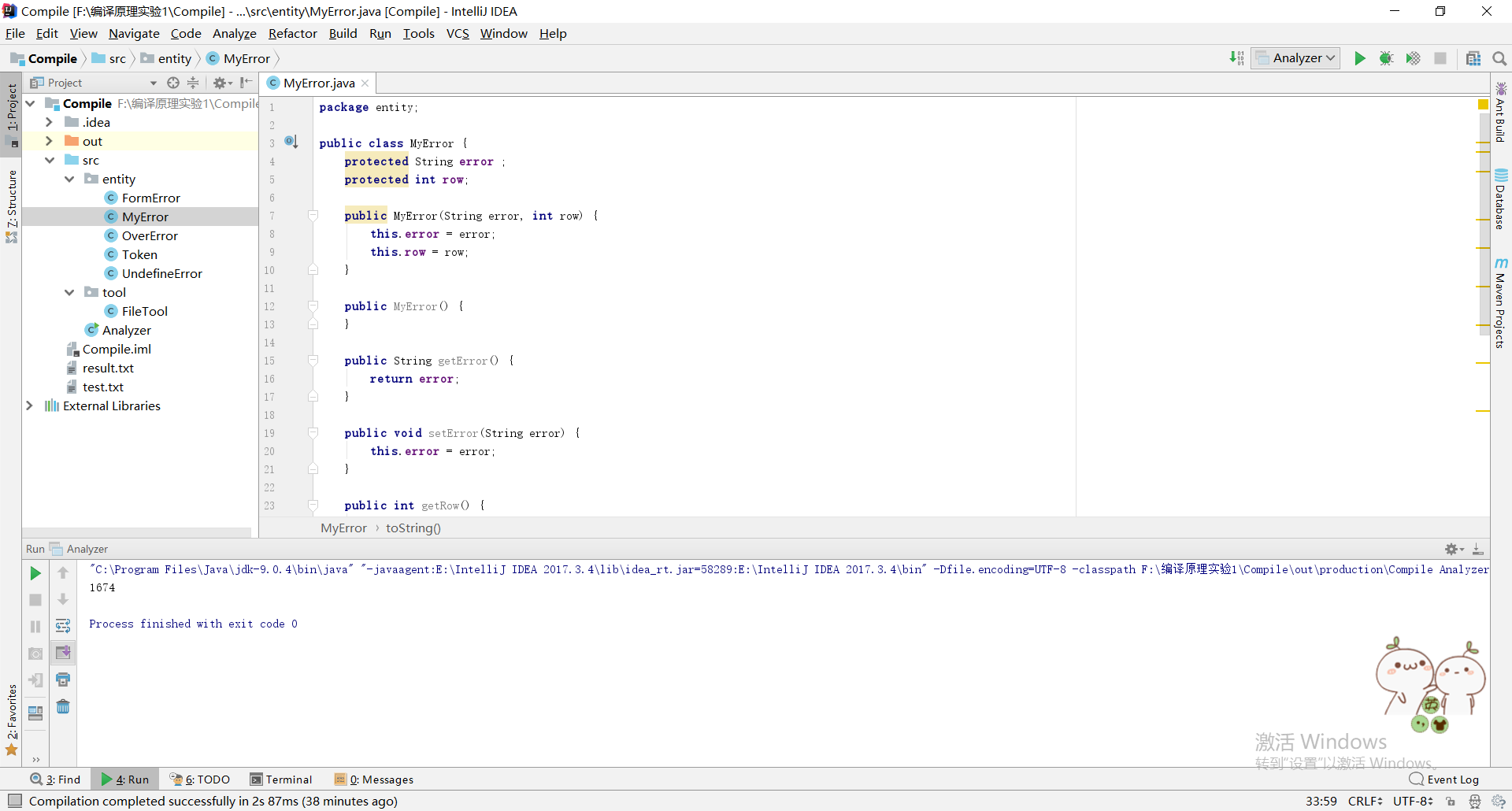
FormError：变量名称格式错误；

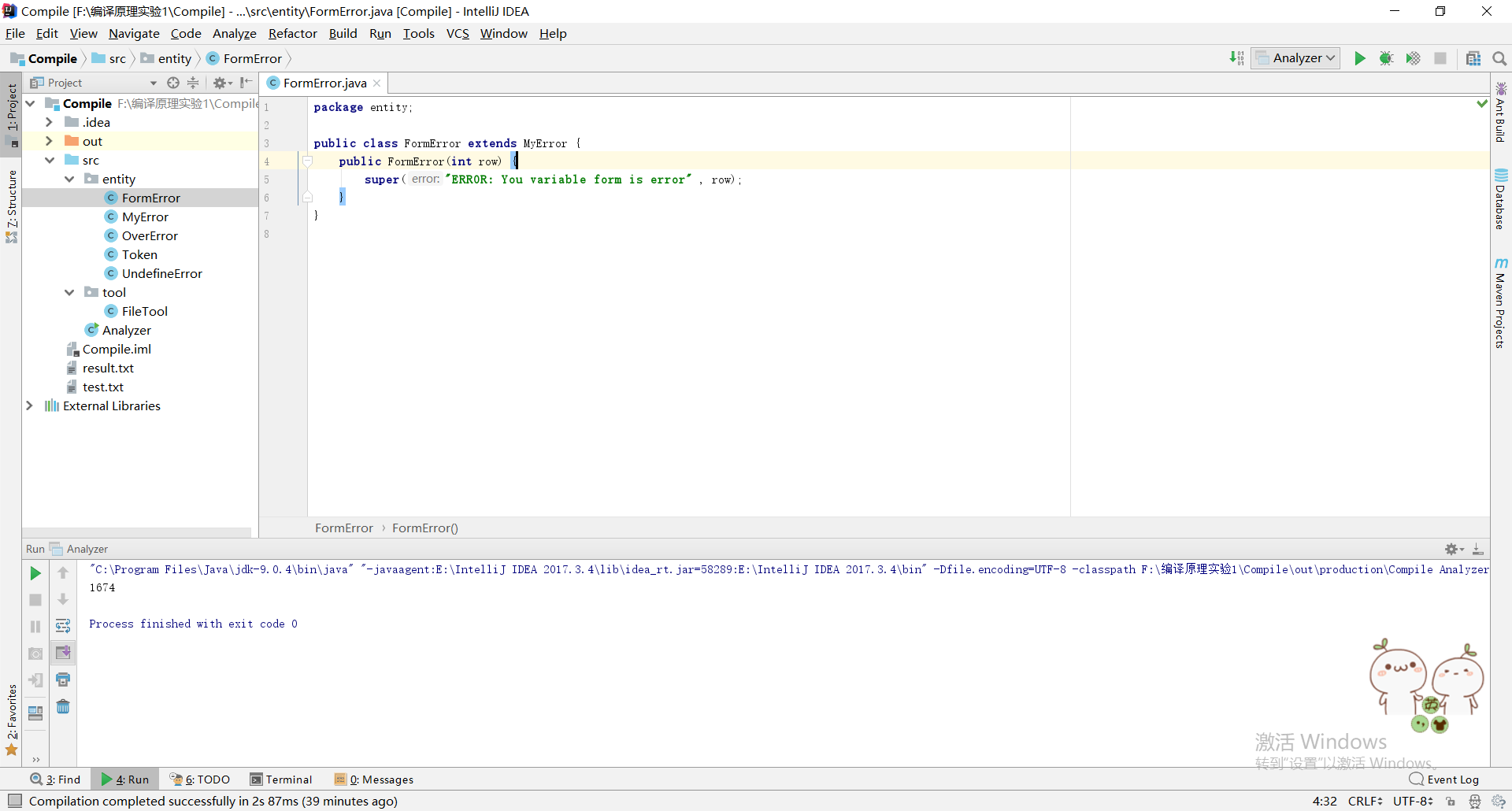
OverError：数字过大错误；

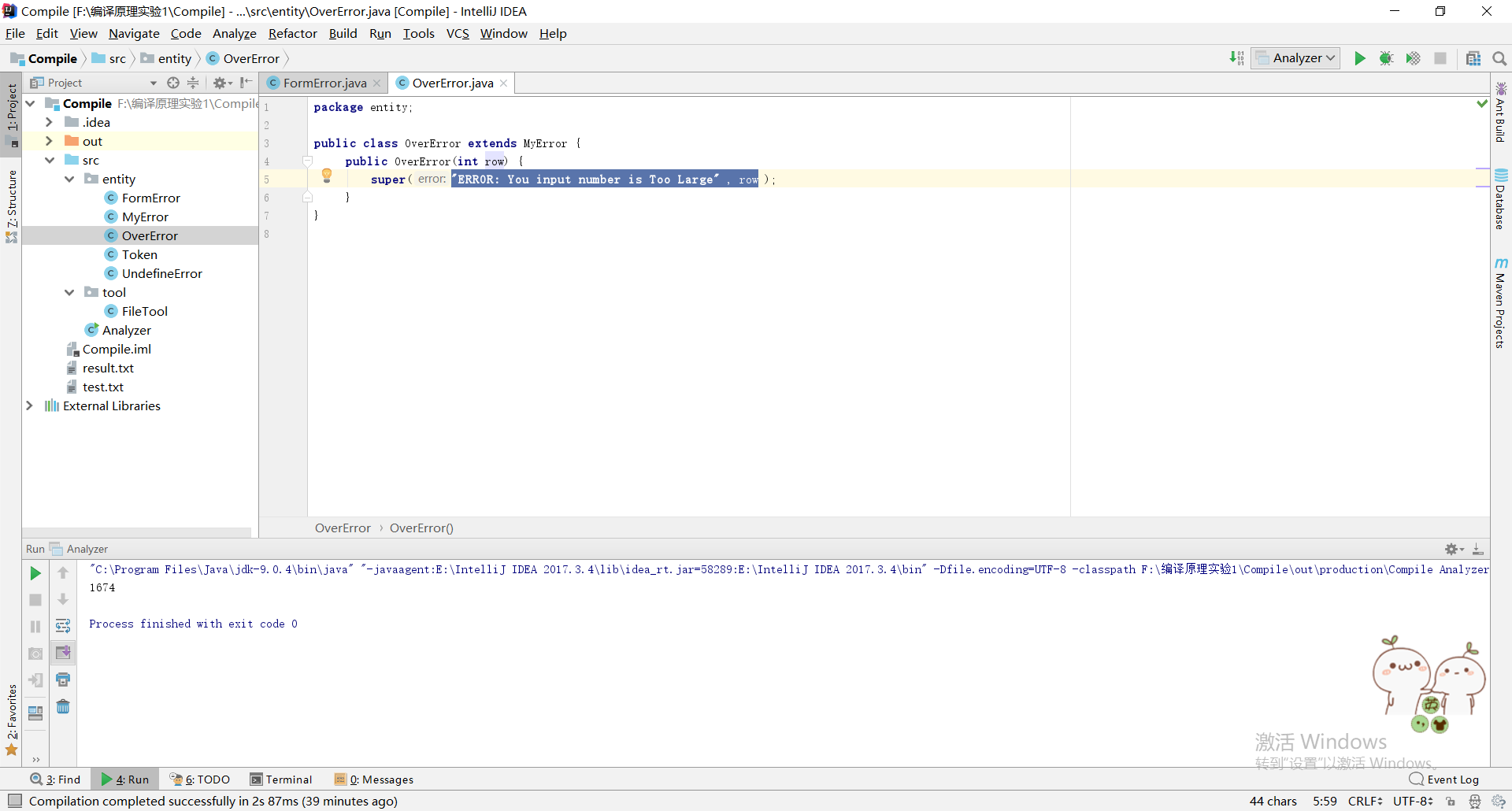
UndefineError：未定义字符错误

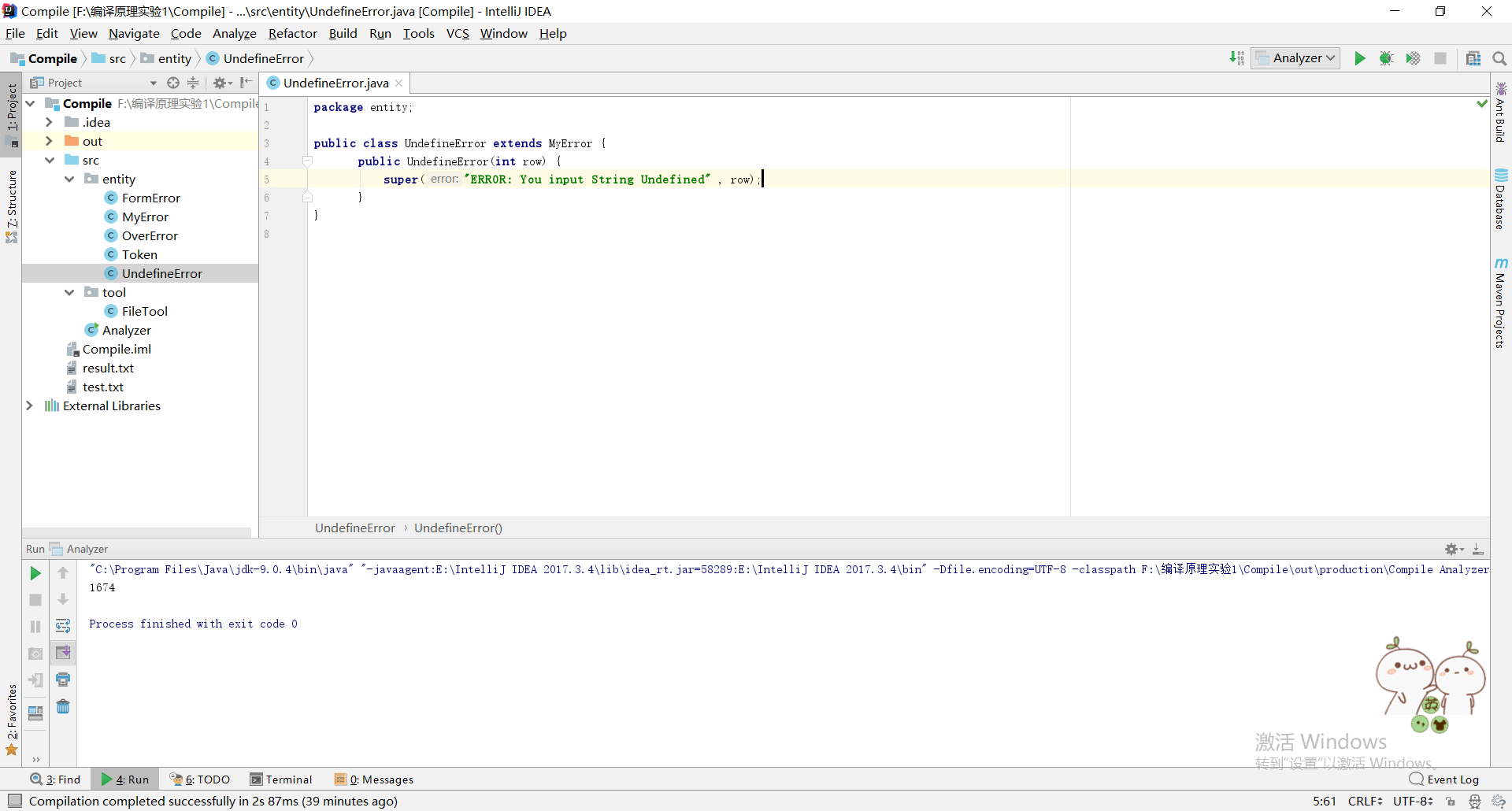
Error属性：错误类型

Row属性：错误行数





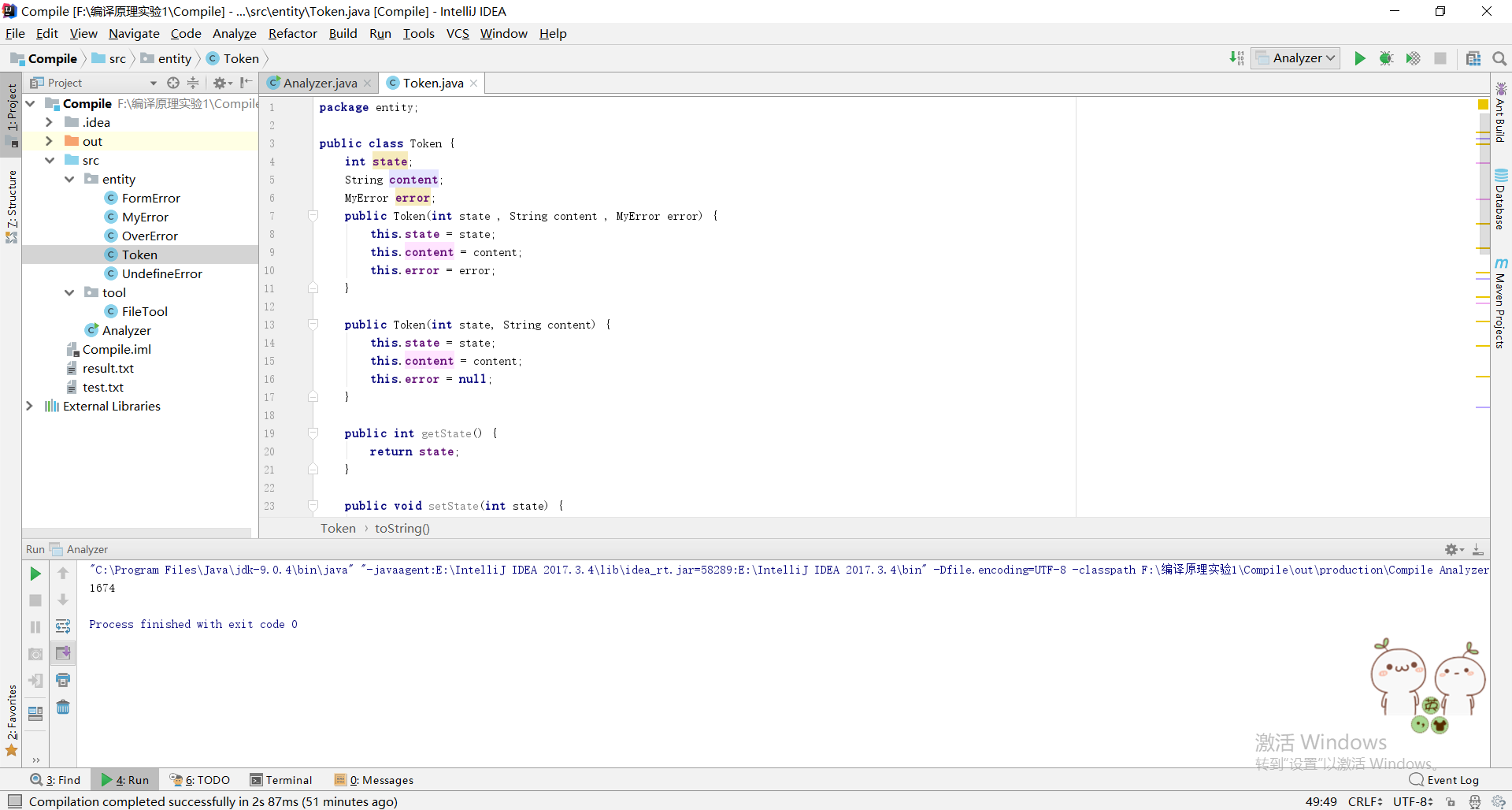




Token类：输出的token序列

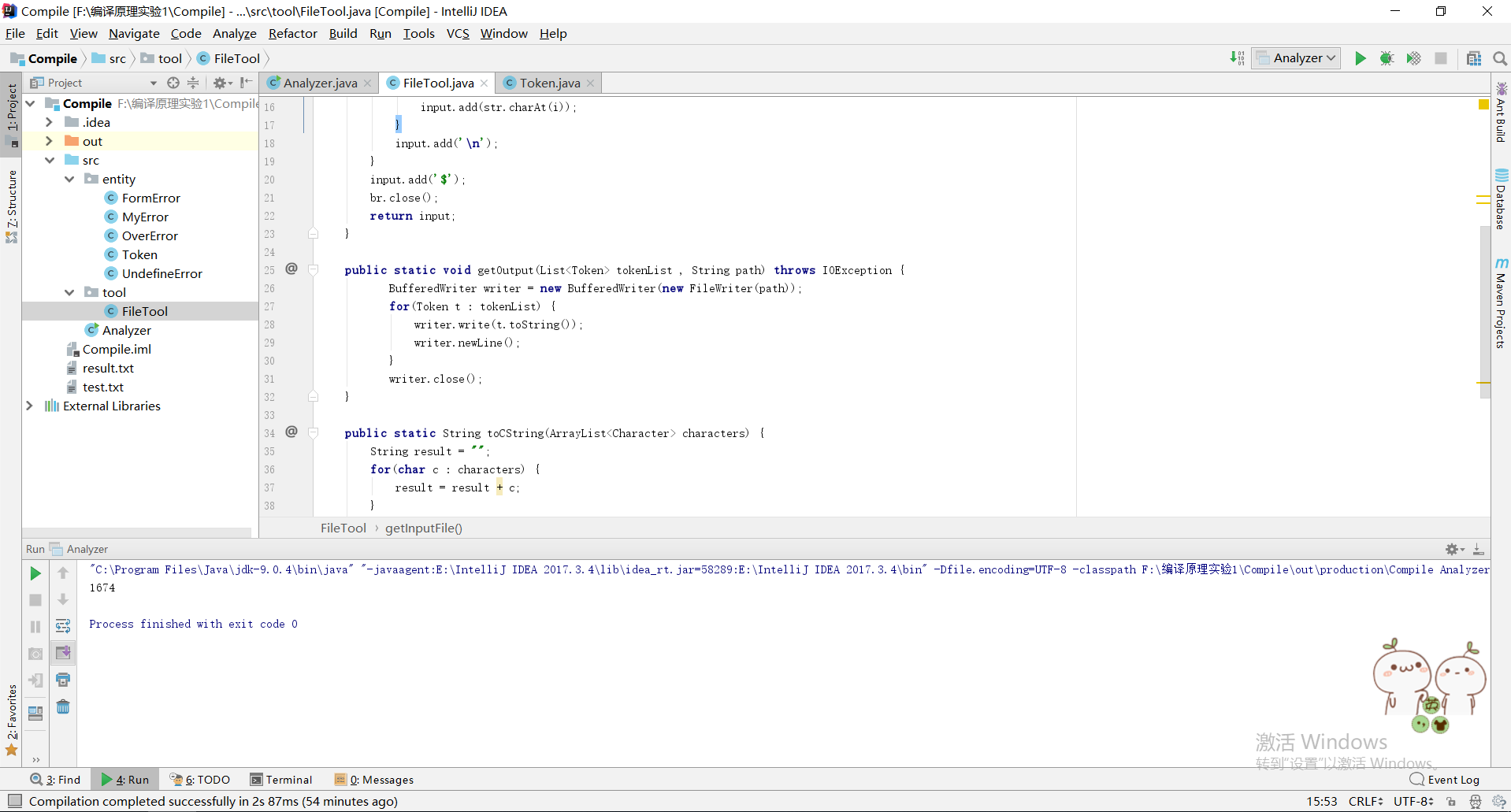
State 和 content对应输出的合法token序列内容。

Error类属性是用来打印错误信息。

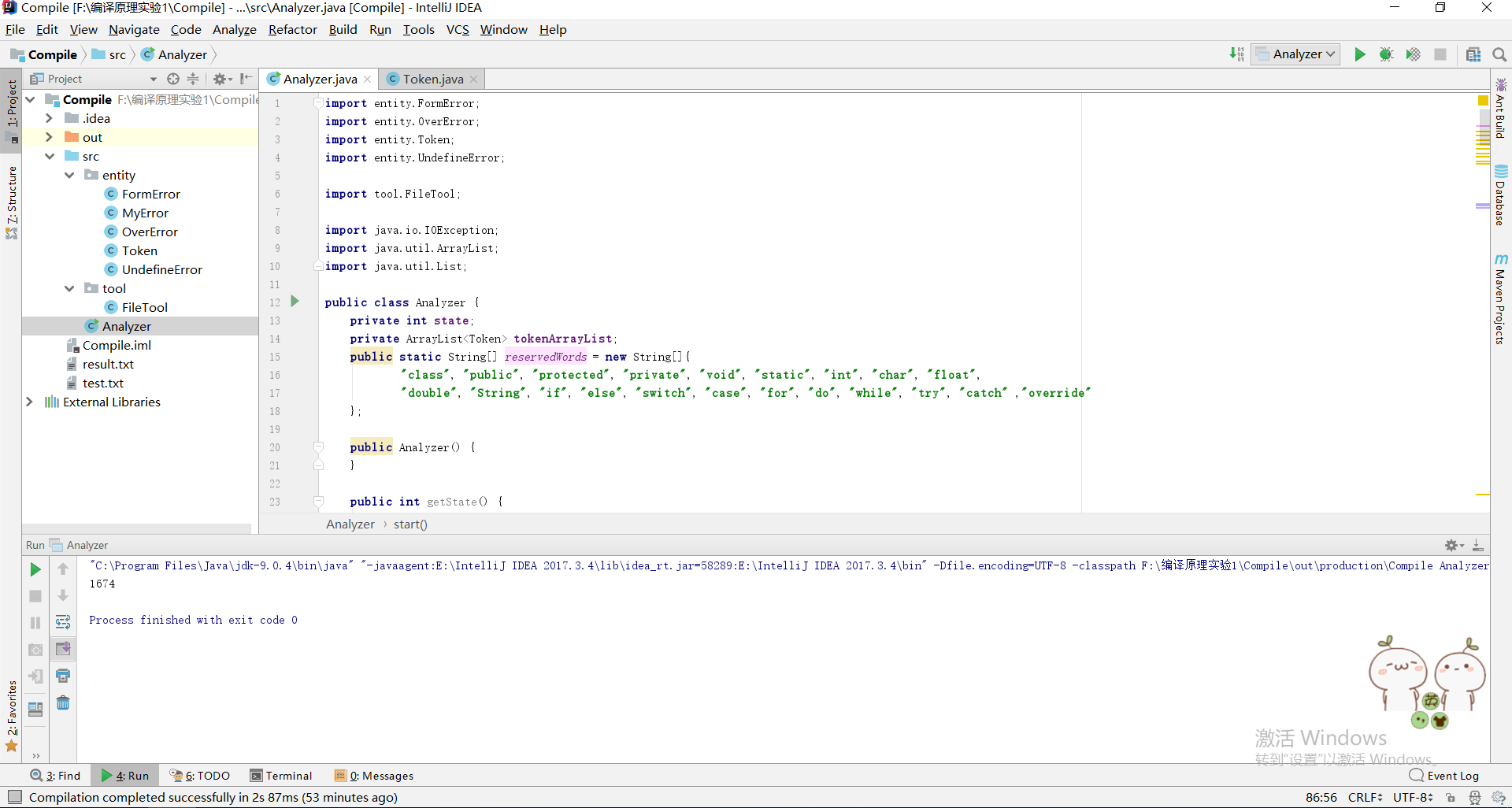


FileTool类：功能:读取文件，输出文件和字符array转字符串

内含三个静态方法。



Analyzer类：实现类

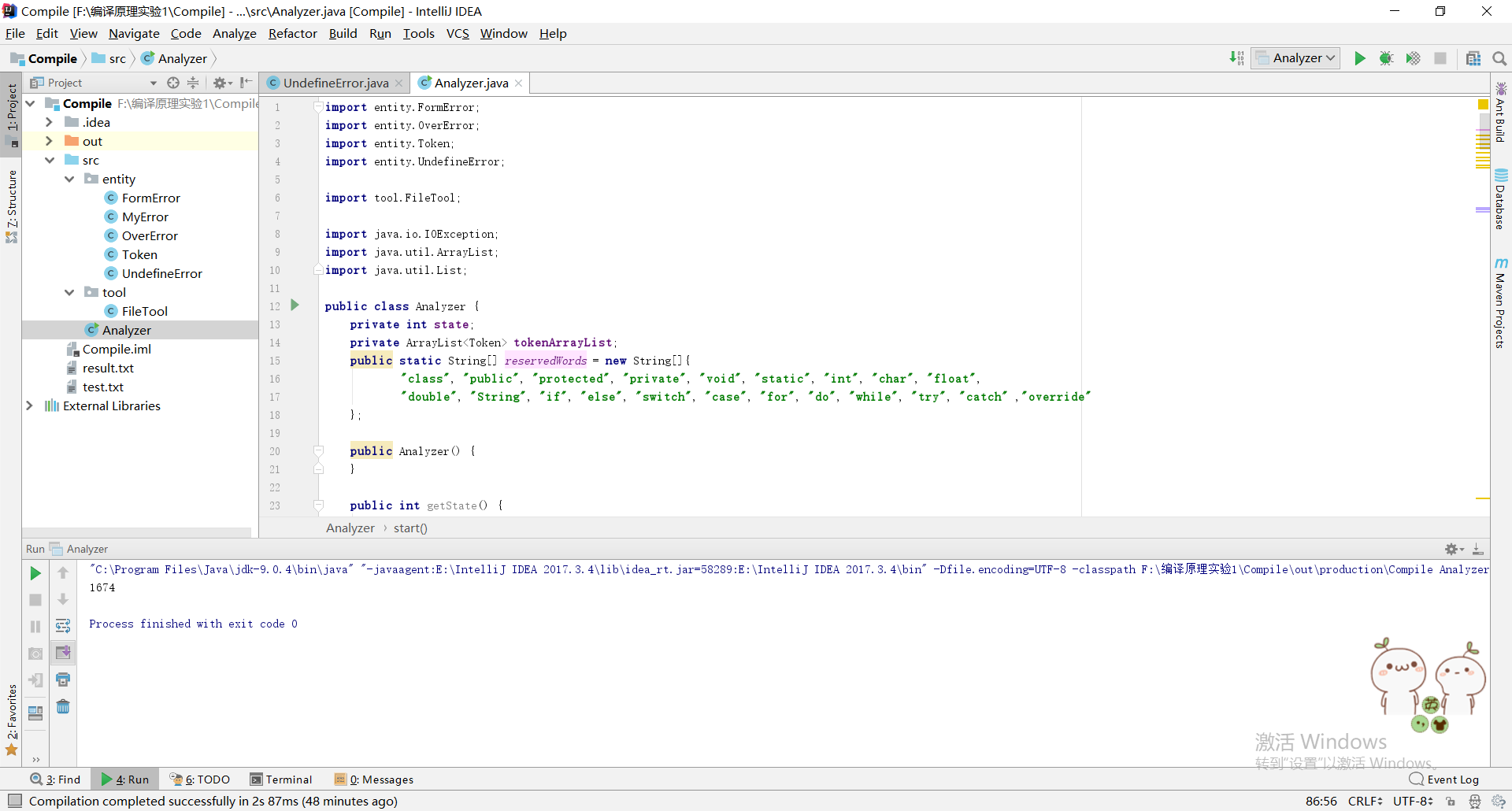


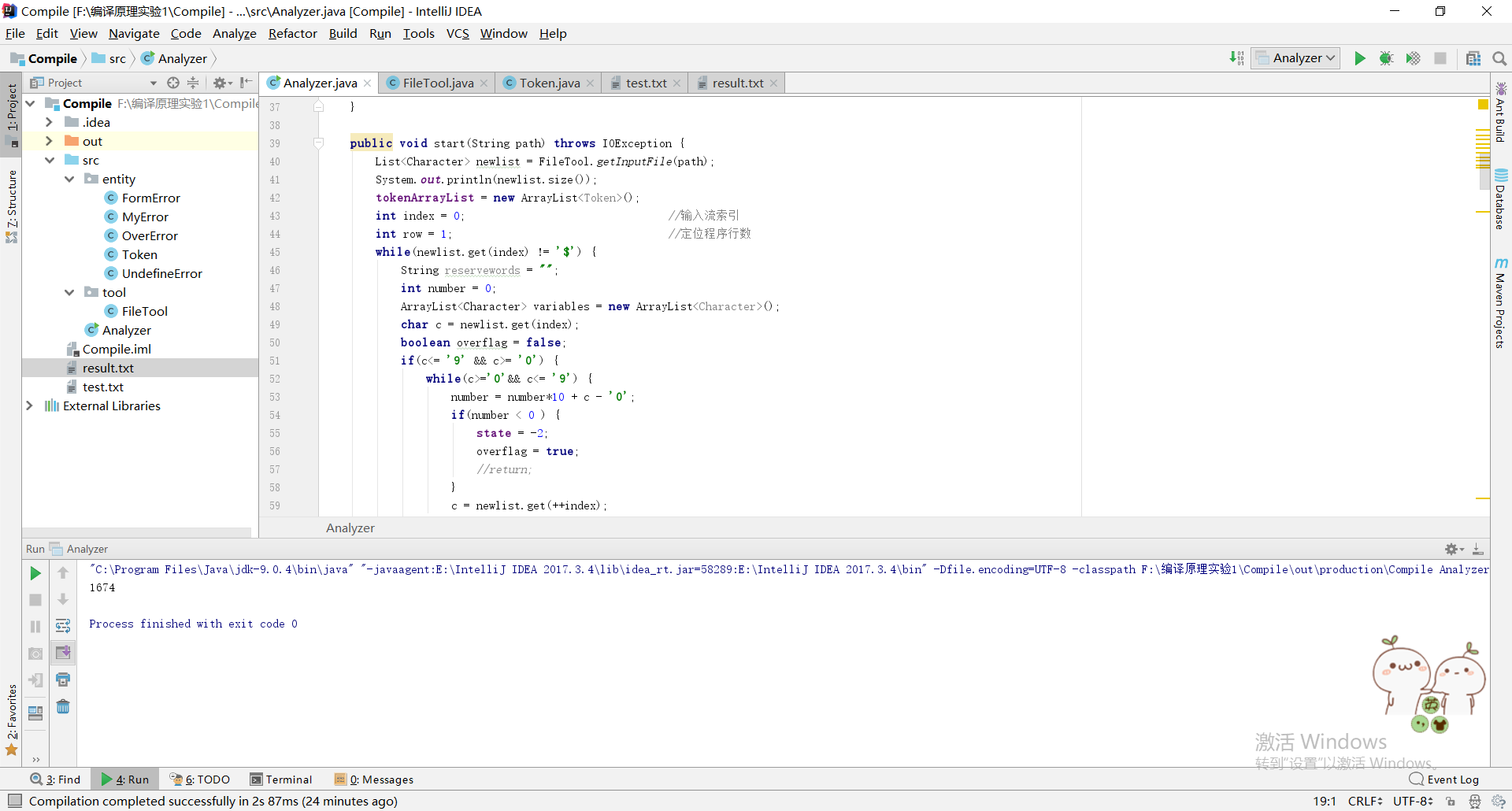
State ：当前分析词法单元的id；

tokeArraylist: 输出的token序列；

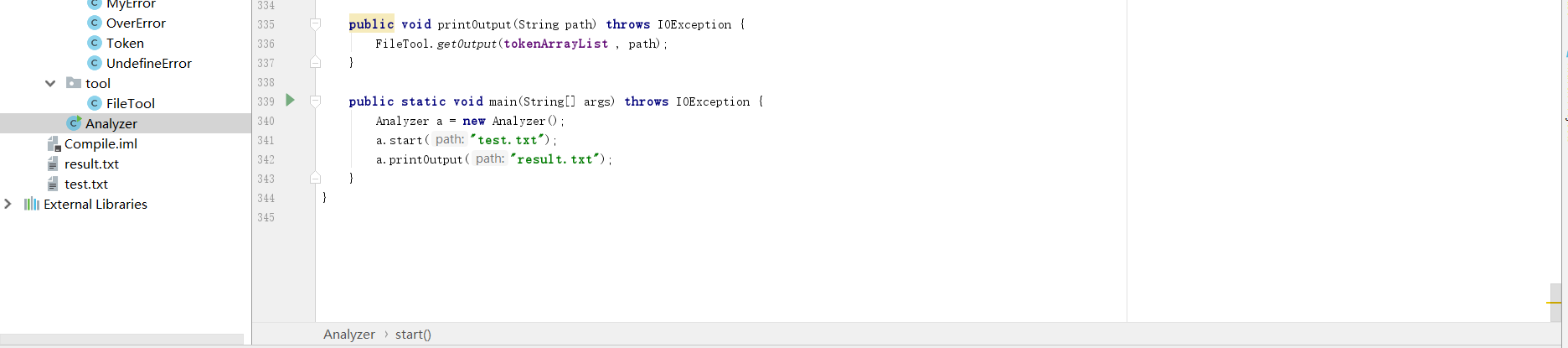
ReservedWords：设定好的静态保留字数组。

# 核心算法



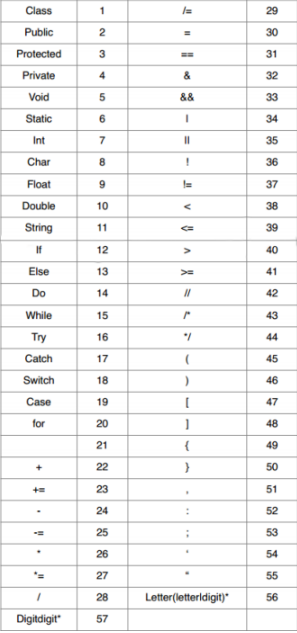


整个实现类是Analyzer类，核心逻辑放在start函数中。输入测试文件的路径path，通过start方法开始分析整个文件，因此该方法逻辑十分复杂。使用while循环，不停的读取字符。通过读取的第⼀个字符的类型，预测接下来的单词符号可能的类型。读到英⽂字符，可能为保留字或变量名（类型⼀）；读到数字，可能是常数（正数）也可能是非法的变量名定义（类型⼆）；读到其他字符（类型三），则可能是操作符或边界符或注释符，如果是‘-’ 符，后⾯是数字就组成了负数，当然也可能是换⾏符或是未定义的字符。 类型⼀，则继续读取，每读⼀位都判断是否属于保留字，若是就直接输出（因为保留字优先于变量名），否则⼀直读到不是英⽂字符为⽌，指针指到下一位，并输出变量名。 类型⼆，⼀直读到不是数字为⽌，并输出正常数，指针指到下一位；若不是正常数字，指针指到下一位，并**报错OverError；**若多读的一位是字符的话，则继续读下去，直到指针不是字符位置，**并报错FormError；**类型三相对复杂⼀点，如果单字符就可以确定种别，则直接输出；否则继续读取下⼀位，直到可以确定种别为止，若负号（减 号）后面是数字，按类型二读取，最后需要输出负数；若不是正常数字，指针指到下一位，并**报错OverError；**如果碰上未定义的字符，指针指到下一位，并**报错UndefinedError。**



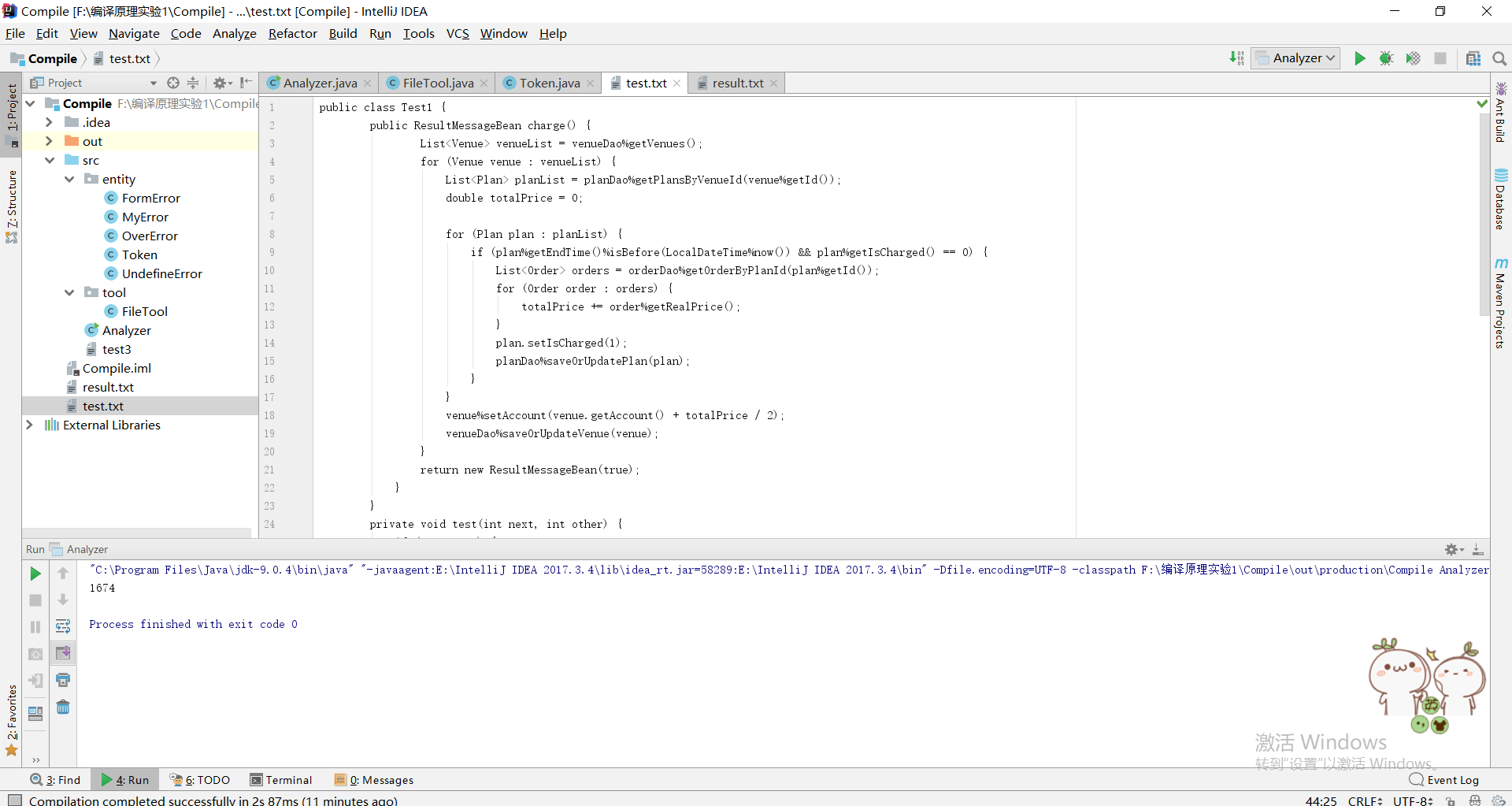
结果通过调用printOutput输出。

**附录：id和content的对应关系**

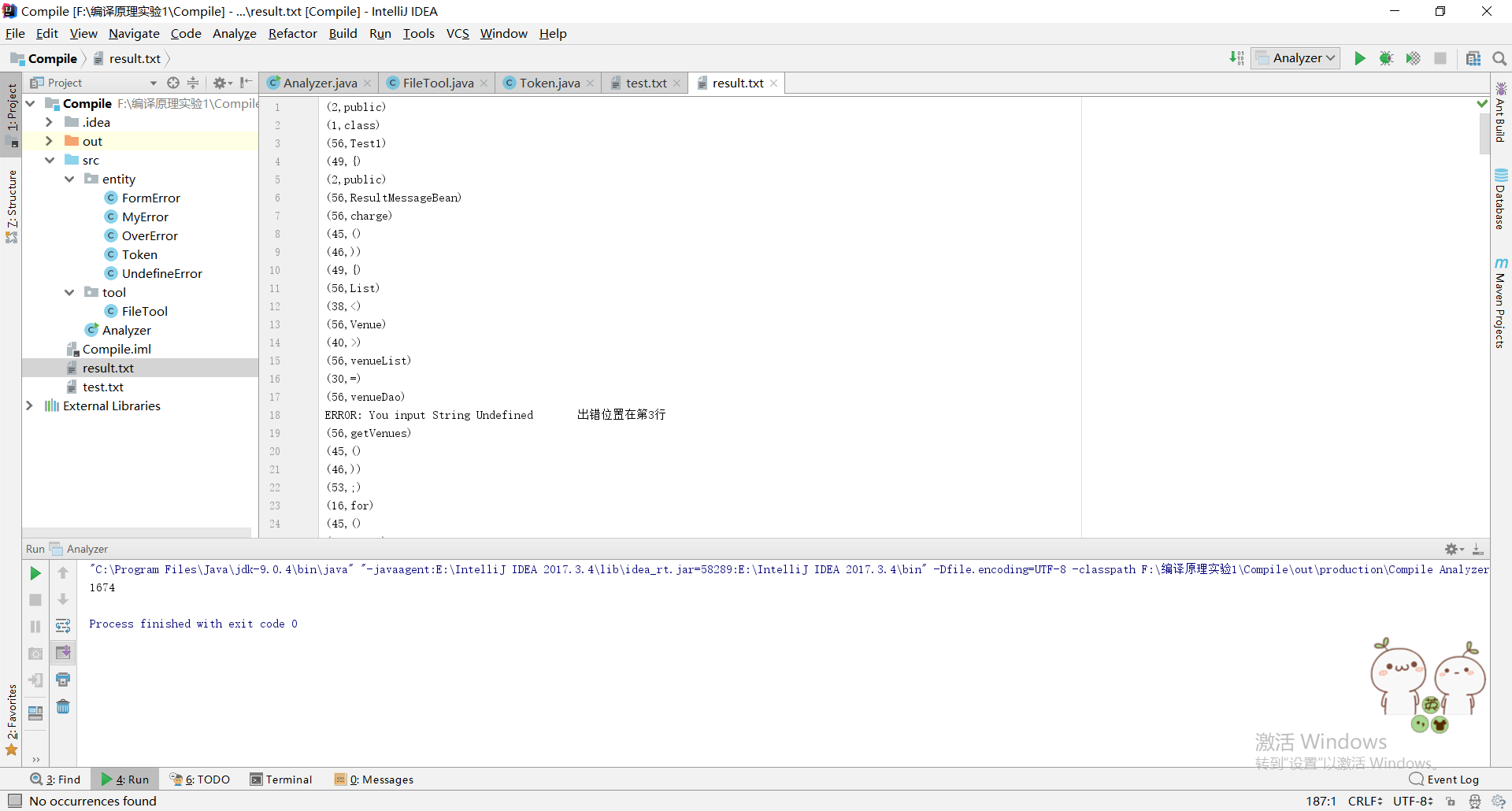


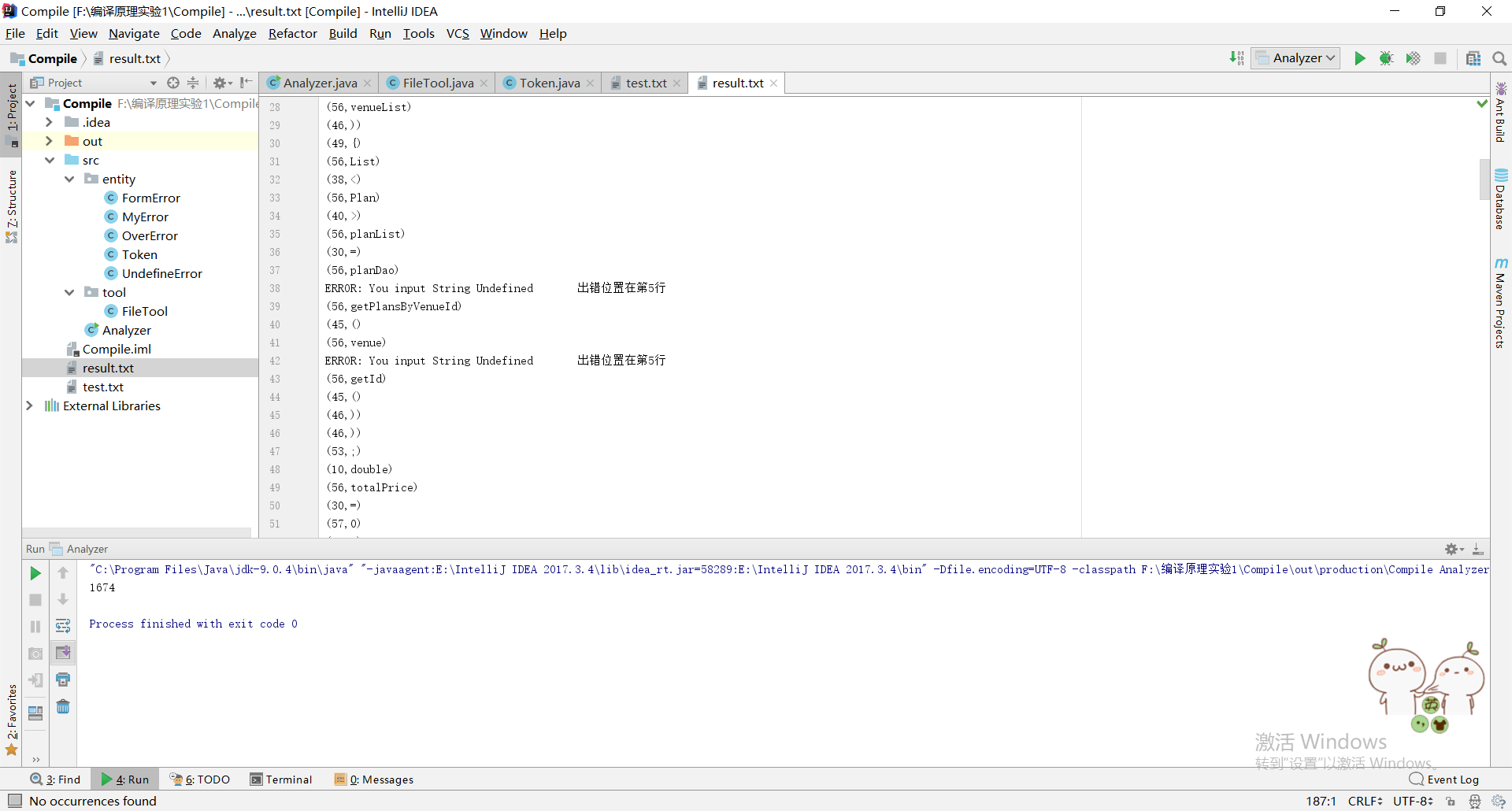
# 运行截图

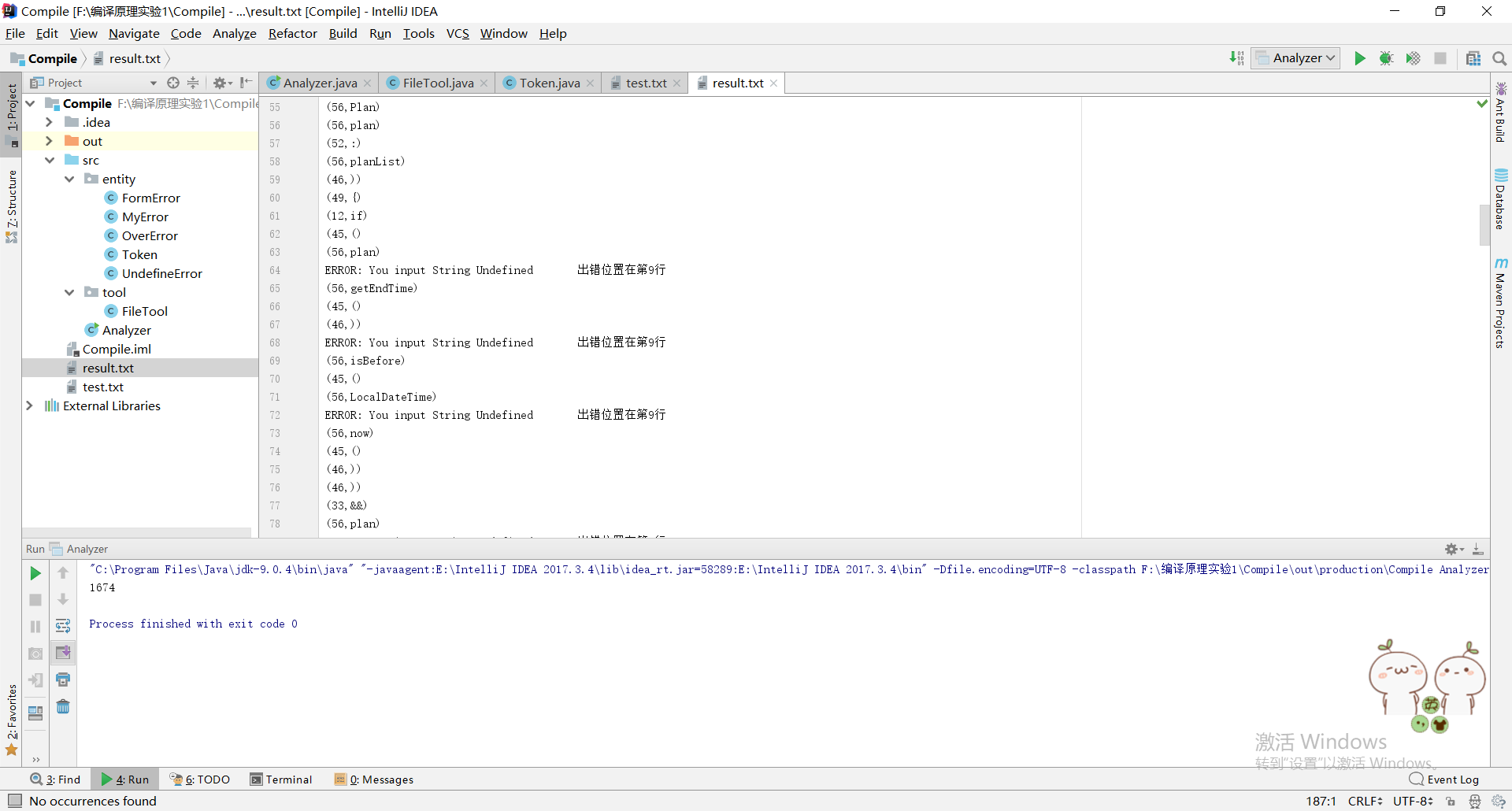
输入test.txt：

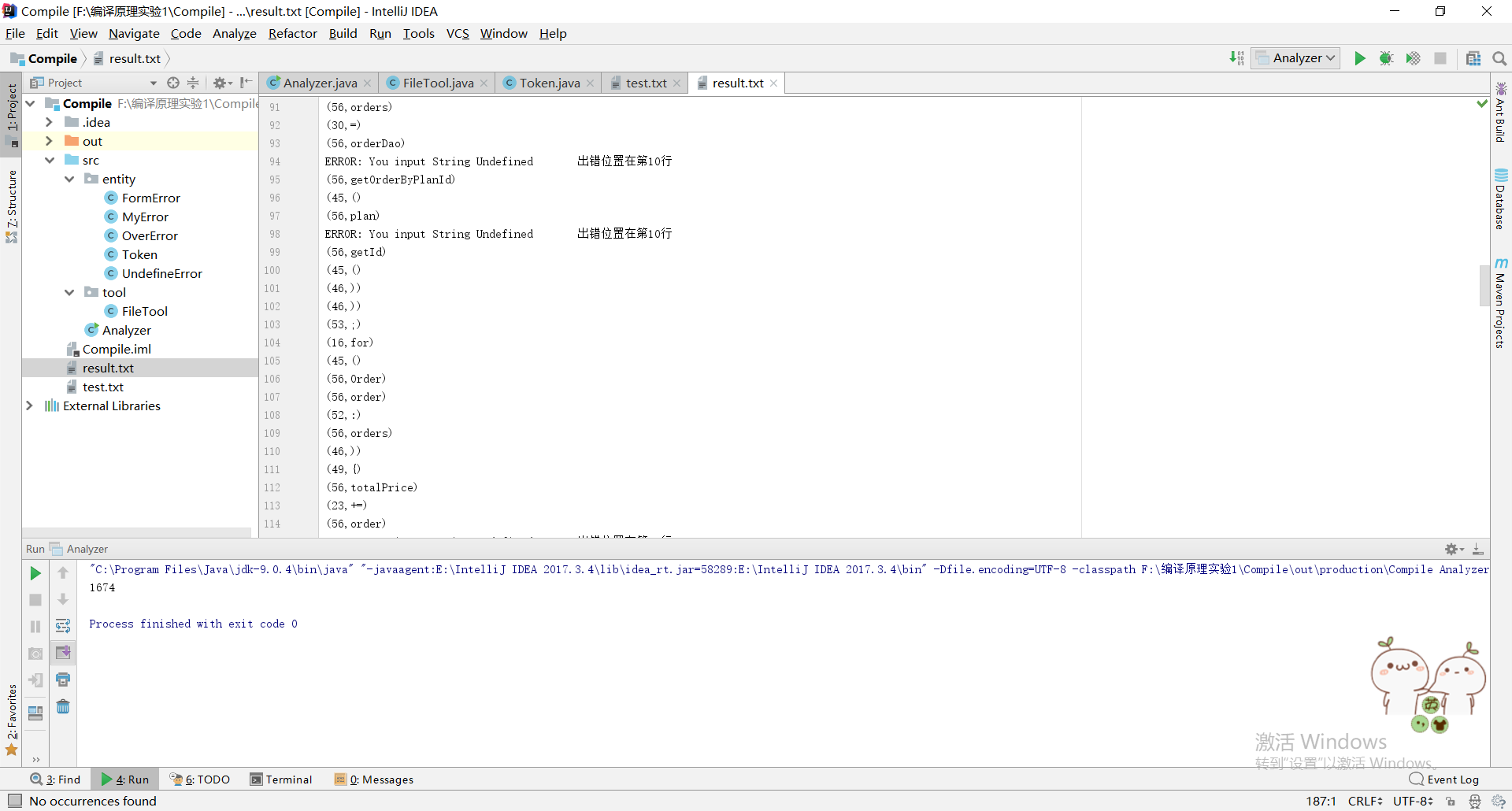


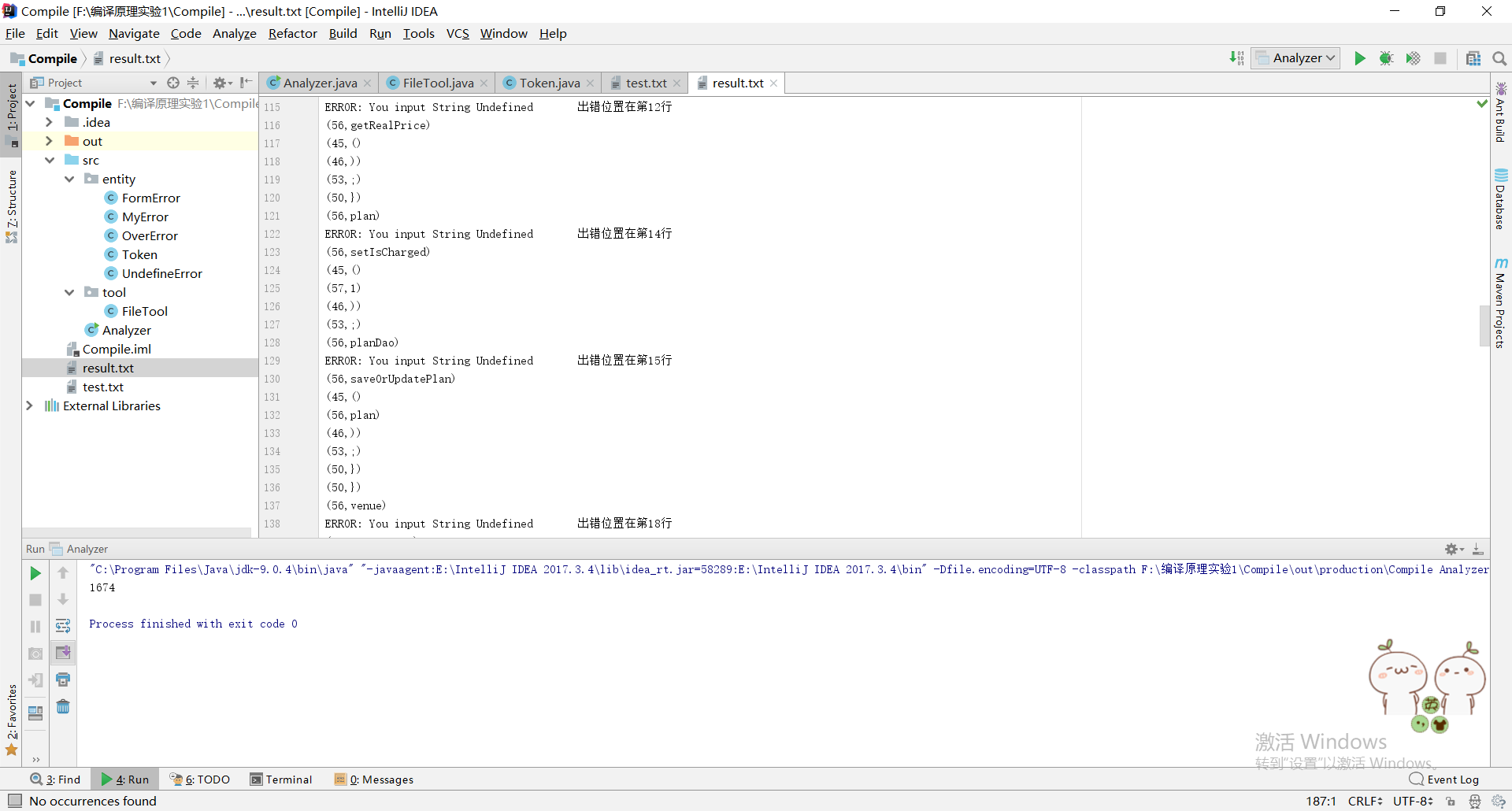
将结果打印到result.txt文件中：

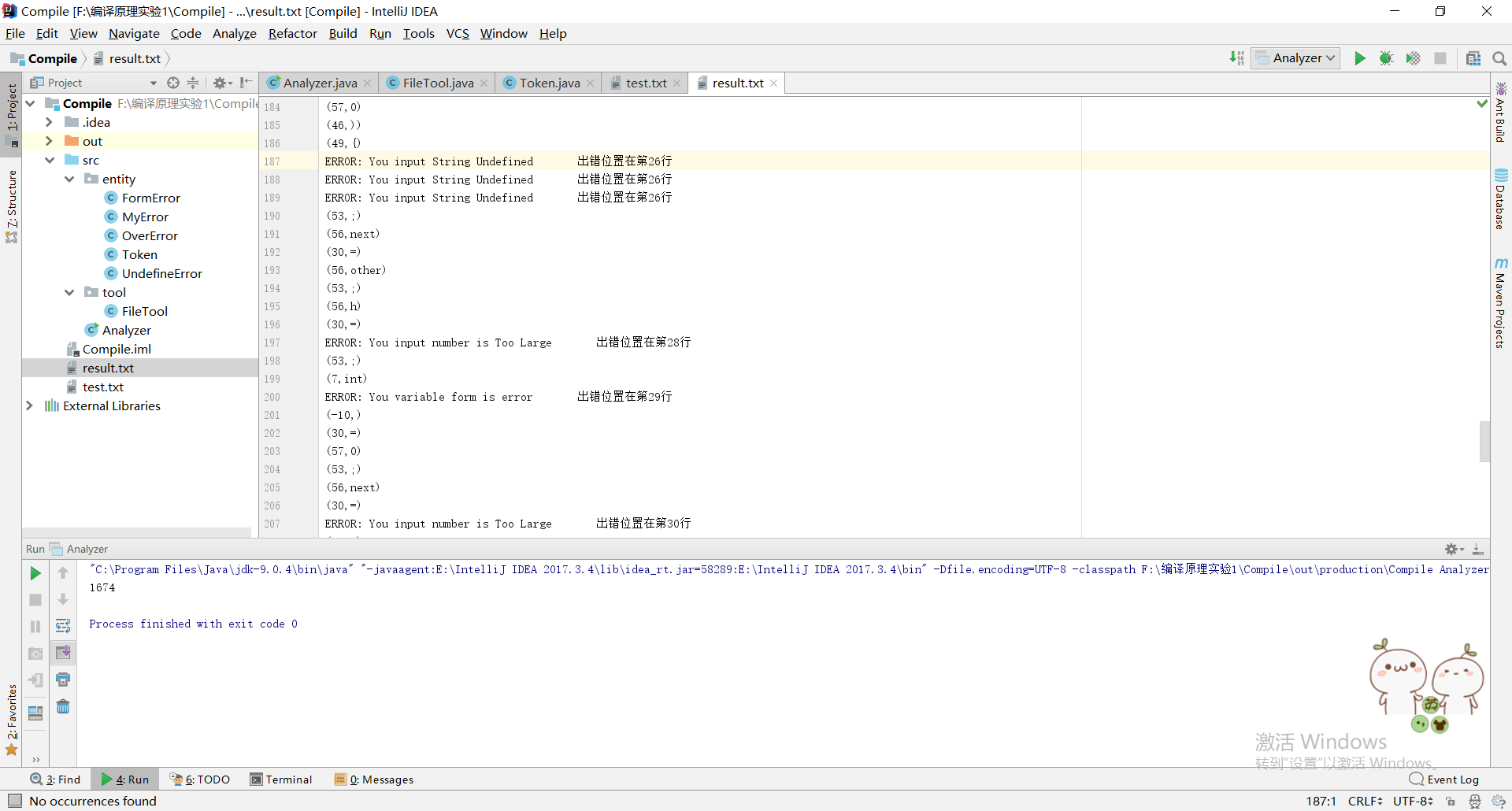




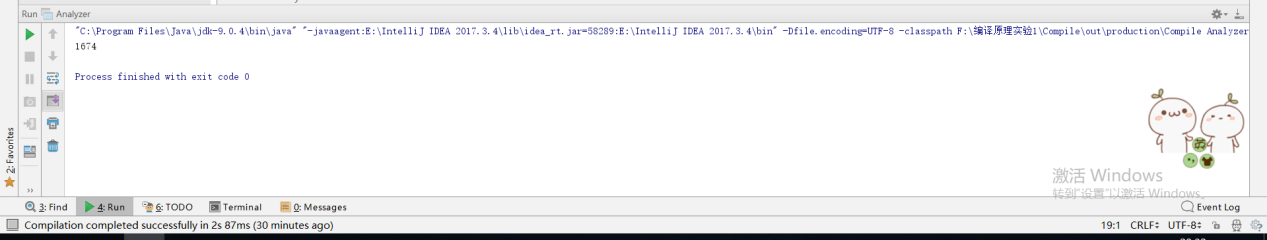








总共分析了1647个词法单元：



# 问题与解决

# 暂无

# 感受与总结

实现这样一个简单的小小编译器，就有着那么复杂的逻辑。让我深刻地感受到要想实现一个识别完备的编译器，是多么的不容易。与此同时，编写词法分析程序，有助于对词法分析过程和⽅法有更深⼊的理解。