中山大學 SUN YAT-SEN UNIVERSITY

编译原理

作业名称	Oberon-0 逆向工程工具 Rose 实验三：自动生成语法分析程序		任课老师	万老师		
院系	软件学院		方向	通信软件	姓名	李明宽
学号	11331173		完成日期	2013 年 12 月 29 日		
QQ	736459905		E-mail	limkuan@mail2.sysu.edu.cn		

实验三：自动生成语法分析程序

一、 实验目的

本实验通过对语法分析程序自动生成工具 JavaCUP 的使用，产生一个 Oberon-0 语言的语法分析和语法制导翻译程序，实现对 Oberon-0 语言绘制函数调用图的功能。通过实现一个语法分析程序加深我们对编译原理中语法分析的理解。

二、 实验过程

1. 下载自动生成工具 JavaCUP

2. 配置和使用 JavaCUP

在这里学习 JavaCUP 的使用方法，参考了 JavaCUP 的官方文档：

http://www.cc.gatech.edu/gvu/people/Faculty/hudson/java_cup/manual.html

参考官方文档的例子渐渐了解 JavaCUP 的使用方法。

3. 生成 Oberon-0 语法分析和语法制导翻译程序

- 由于语法分析也是建立在完成了词法分析的基础之上，因此也需要使用 JFlex 生成一个词法分析程序，它所使用到的终结符通过 CUP 文件定义。
- 在 JavaCUP 的代码文件中 init with 后面接的是语法分析开始之前的初始化代码，一般情况可以用来初始化各种全局变量以及语法分析工具等等。
- scan with 后面接的是获取到下一个 Token 所调用的函数，通常情况来说都是 getScanner().next_token();
- parser code 后面接的代码是会出现在语法分析程序中类的定义里。
- 接下来声明终结符以及非终结符，符号文件 Symbol.java 是 JavaCUP 自动生成，里面包含了在这里定义的所有的终结符。非终结符是接下来进行文法推导的时候需要使用到的。
- Precedence 里声明了部分运算符的优先级，防止文法出现的二义性。
- 接下来就定义了 Oberon-0 语言的文法了，通过参考 Oberon-0 语言的 EBNF 定义，然后把[]{}等符号展开成多个文法推导式，完成一个能识别 Oberon-0 语言的语法分析程序。
- 定义对每一个文法推导式的语法制导翻译语句，实现异常处理以及调用图的绘制。

4. 讨论不同生成工具的差异

- Bison 中配套使用的是 Lex 词法分析程序生成工具，而 JavaCUP 使用的是 JFlex。Bison 主要是基于 C 语言实现的语法分析程序，JavaCUP 是基于 Java 语言实现的，因此在生成的代码中会有很大的不同，JavaCUP 利用了很多 Java 的特性如类。
- 在整个生成程序结构方面，JAVA 的程序结构如下左图所示，Bison 的程序结构与 Lex 类的词法分析工具类似，如下右图所示：

Package 和 import 声明 用户初始化代码 终结符与非终结符的声明 定义文法规则

终结符与非终结符的声明 %% 定义文法规则 %% 用户自定义代码

- 在语法制导翻译上，JavaCUP 通过在非终结符或终结符后面加上冒号和标识符来声明这个符号，而 Bison 是按顺序通过${num}来获取到符号的返回值。对于非终结符的返回值，JavaCUP 使用的是 RESULT 变量来进行返回，而 Bison 中使用$$返回类型。
- 在这个实验中由于使用的是 JavaCUP 语法分析程序生成工具，而对 Bison 中 Yacc 了解不是很多，不过感觉如果通过 Yacc 来完成实验，步骤应该也差不多。

三、 关键问题及解决办法

1. 已实现的功能

1) 函数调用图的绘制

在 Procedure 过程声明的时候调用 graph.addProcedure()往图中添加一个调用目标，然后在所有语句中，如果匹配了 call_procedure，就通过 graph.addCallSite()添加一个调用者并且使用 graph.addEdge()添加一条从调用者指向被调用者的箭头。

为了能够使得程序中能够调用那些在所在函数出现之后声明的函数，因此我使用一个 HashMap 保存了需要添加的调用者以及调用关系，在结束的时候才把这些连线绘制并通过 show 函数把函数调用图显示出来。

对于预定义的函数 read,write 和 writeln，需要在 init code 中手工添加方便以后调用。

2) 类型不匹配异常的判断及处理

使用 Variable 类来记录类型以及对应的值，然后使用 HashMap<String, Variable> 来记录 Oberon-0 语言中出现变量名以及他的类型，在进行一些赋值操作，if/while 判断语句，表达式中求值等操作的时候判断拿到的 Identifier 的类型，对比他们的 Type 如果不符合要求的话，就抛出异常。

比较麻烦的是函数调用过程中出现的类型不匹配。由于我是用的是 HashMap 来存储在过程声明 heading 中出现的变量名以及它的类型定义，这样遍历 Map 会导致参数顺序的变化，这样判断相应的类型不匹配错误就会出现错误。我使用的办法是存函数参数名的时候通过字符串划分加上它所在的序号，然后需要的时候使用 split 得到序号或者变量名，这样来对类型进行检查。

3) 变量作用域的使用规则

使用一个 stack 来存储各个作用域中的变量，当识别到 declarations 的时候把这里定义的 HashMap 也就是变量集合压入栈中，并把在本层没有定义但在上一层定义了的变量名压入到变量的集合，这样栈顶中所存的 HashMap 就是当前所能用的所有变量名，然后在作用域结束的时候，例如一个 Procedure 过程声明结束的时候把栈顶元素弹出，这样来说如果一个变量名在栈顶的 HashMap 中不存在的话，那就是一个未定义的变量。

4) 函数调用中参数个数不匹配异常

使用一个 ArrayList 记录了函数调用语句中出现的参数，然后对比相应的存储函数参数的 HashMap 的元素个数，如果不一样，那就是参数个数不匹配。

5) 常量表达式的求值

重写了 expression、term、factor 等文法推导，然后在制导翻译动作里完成计算，有个缺陷就是由于变量的赋值没有处理好因此没实现对变量的求值。对于一个整型的变量使用 0 参与运算，一个布尔型的变量使用 false 参与运算。

6) 异常抛出时记录问题的位置

通过在词法分析程序中封装 yyline 以及 yycolumn，然后在抛出异常的时候调用 getScanner()还要强制类型转换成 OberonScanner，然后里面的函数 getLine 以及 getColumn 获取到错误发生的位置。

2. 有待后续实现的功能

1) 赋值语句以及变量表达式求值

这是由于 selector 返回值是 Variable 导致，这个应该比较好解决，只要把 selector 的返回值变成 HashMap<String, Variable>，然后记录 Identifier 变量名，这样就能够通过栈顶的 HashMap 获取到并修改当前 Identifier 的值并且完成赋值操作。理论上说完成了赋值语句的使用后变量的表达式求值应该也能实现了。

由于时间问题这部分就没有实现……

2) 部分缺失括号，缺失分号缺失操作符，运算数的异常处理

我本来想直接通过文法推导式的匹配来完成这些异常处理，但是发现部分可以，但一些如括号的缺失的文法推导式或者操作符缺少的推导式，会导致移入规约冲突！只能放弃这种办法。别的实现方法没时间想了 T_T

希望这方面的异常处理在接下来的过程中能够实现。

3) 数组类型嵌套处理

这个真心想了好久，对于嵌套的多维数组作为函数参数调用的时候，无法判断是否这个类型是否匹配，因为在现在的实现方法中，无论多少维的数组存储的类型都是最终数组中元素的类型，而且对于多维数组，获取到相应的数组元素的值比较困难，要在 Java 中开辟一个类似的多维数组存储空间，比较麻烦而且开辟的时候类型还不是确定的 @o@”我觉得后期如果要解决这个问题，一种很可能的方式是改变当前存储变量类型的模式，让它能够对数组特殊处理，或者改变有关数组定义那部分的文法，使得在生成数组的时候已知数组元素的类型，总感觉这个在短时间内搞不定……

4) 测试样例的问题

由于写第一部分的时候没有很仔细研究文法，因此写出来的测试样例有语法上的错误，（这个我会修改的！）这次的测试样例我是用的是软装置中提供的测试样例，除了语法错误有较多的判断不出，词法以及语义上的错误基本能处理。

5) 选择操作进行了简化

为了方便处理，我在文法中限制了选择操作的处理，只能通过 id1.id2 这种方式对 Record 进行调用，也就是说 Record 不允许嵌套也不允许出现 Record 数组……这样感觉不太好，但修改的时间已经不足……就留到接下来的工作解决吧……

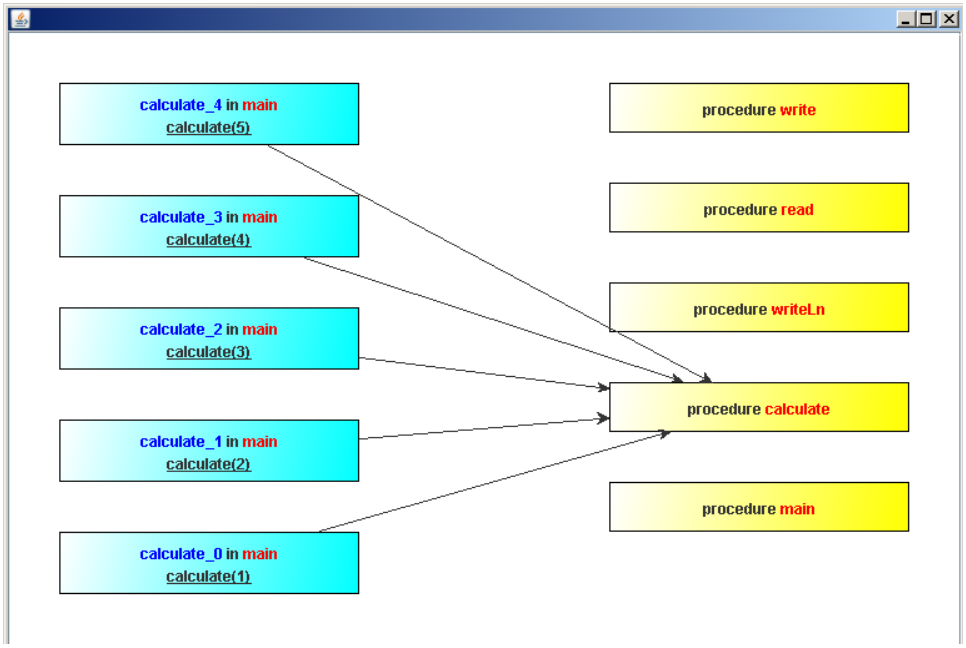
四、实验成果展示

1. 目录结构与文件说明

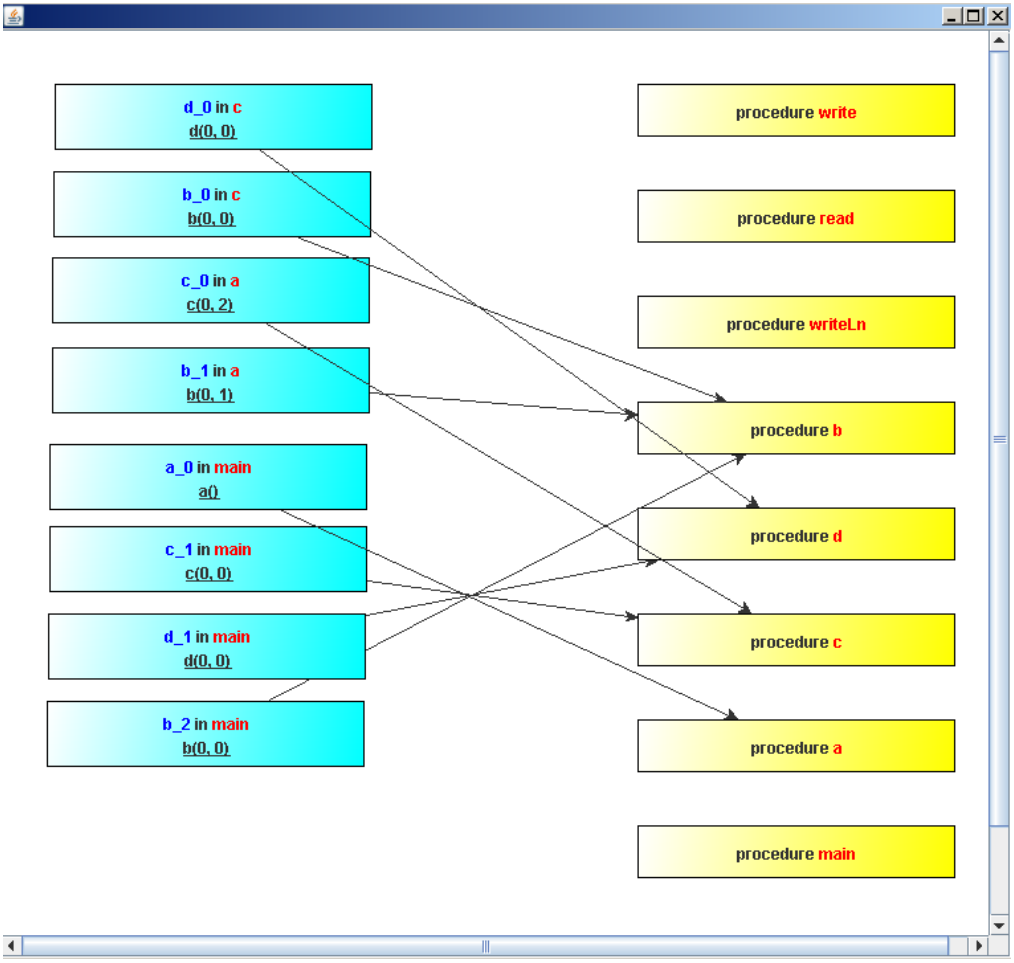
```
➜  ex3  tree
.
├── bin……编译输出的字节码文件
│   ├── CUP$Parser$actions.class
│   ├── …
│   └── Variable.class
├── doc……生成的 JavaDoc 文档
│   ├── allclasses-frame.html
│   ├── …
│   └── Variable.html
├── javacup……使用的 JavaCUP 工具
│   └── java-cup-11a.jar
├── lib……程序中使用到的类库
│   ├── callgraph.jar
│   ├── flowchart.jar
│   ├── java-cup-11a.jar
│   ├── JFlex.jar
│   └── jgraph.jar
├── src……源代码文件夹
│   ├── exceptions……所有异常类
│   │   ├── DivideByZeroException.java……被零除异常
│   │   ├── …
│   │   └── TypeMismatchedException.java……类型不匹配异常
│   ├── IdType.java……定义了语言中变量的类型
│   ├── oberon.cup……JavaCUP 输入文件
│   ├── oberon.flex……JFlex 词法分析生成源文件
│   ├── OberonMain.java……Oberon 语法分析主程序
│   ├── OberonScanner.java……JFlex 生成词法分析程序
│   ├── Parser.java……JavaCUP 生成语法分析程序
│   ├── Symbol.java……JavaCUP 生成终结符表
│   ├── testcases……测试样例（使用软装置中提供）
│   │   ├── Factorial.obr……正确的源程序
│   │   ├── LexicalErrors……词法错误变异程序
│   │   │   ├── Test.001
│   │   │   └── …
│   │   ├── Sample.obr……正确的测试样例
│   │   ├── SemanticErrors……语义错误变异程序
│   │   │   ├── Factorial.001
│   │   │   └── …
│   │   ├── Sort.obr……正确的测试样例
│   │   └── SyntacticErrors……语法错误变异程序
│   │       ├── Factorial.001
│   │       └── …
│   └── Variable.java……存储 Oberon 变量的类
├── test_lexical_error.bat……测试词法错误变异程序脚本
├── test_semantic_error.bat……测试语义错误变异程序脚本
├── test_syntactic_error.bat……测试语法错误变异程序脚本
├── build.bat……构建脚本
├── clean.bat……清理项目脚本
├── run.bat……运行正确的测试样例脚本
├── doc.bat……生成 JavaDoc 脚本
├── gen.bat……生成词法语法分析源程序脚本
├── readme.txt……说明文件
├── yaccgen.pdf……实验报告 PDF 格式
└── yaccgen.doc……实验报告 DOC 格式
```

2. 程序运行截图：

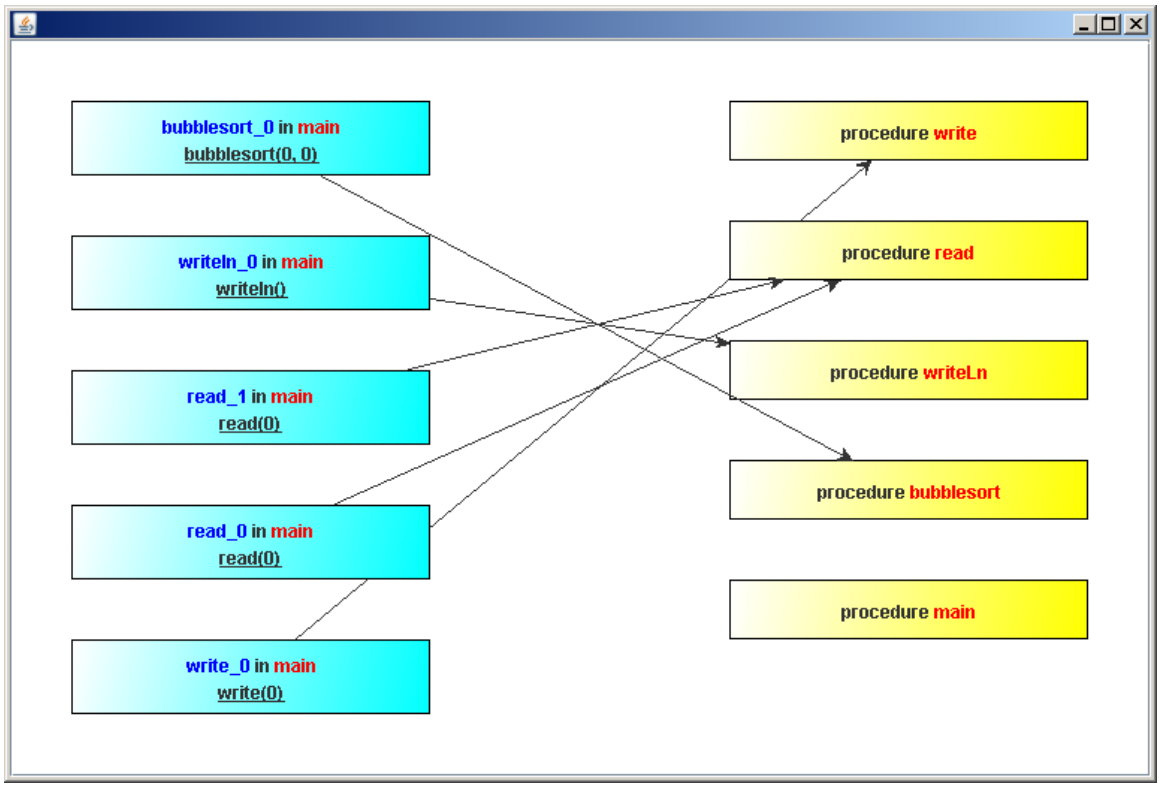
➢ 运行 Run.bat 脚本分析 factorial.obr 源程序结果：



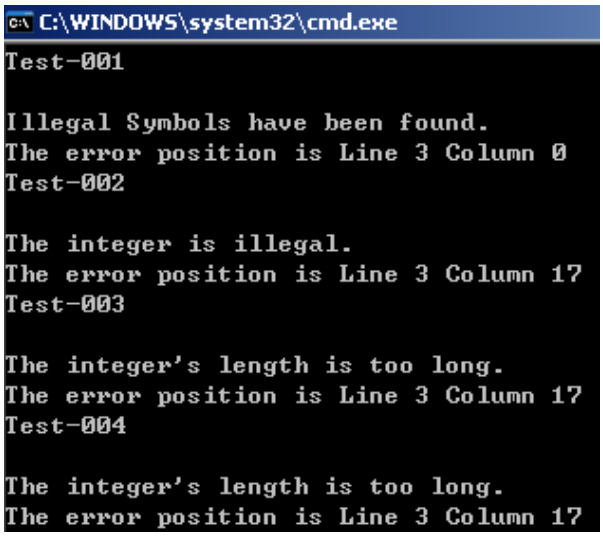
➢ 运行 Run.bat 脚本分析 callgraph.obr 源程序结果：

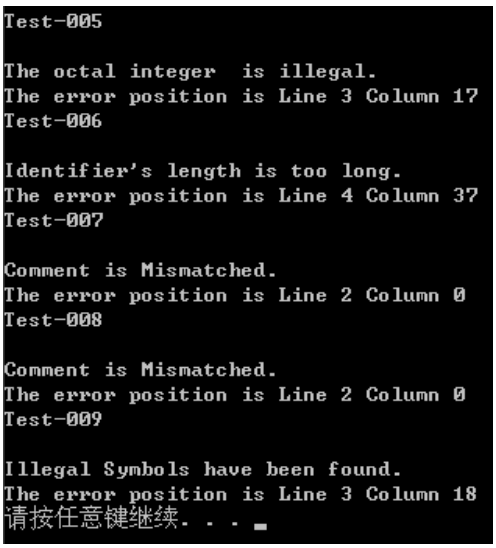


- 运行 Run.bat 脚本分析 sort.obr 源程序结果：

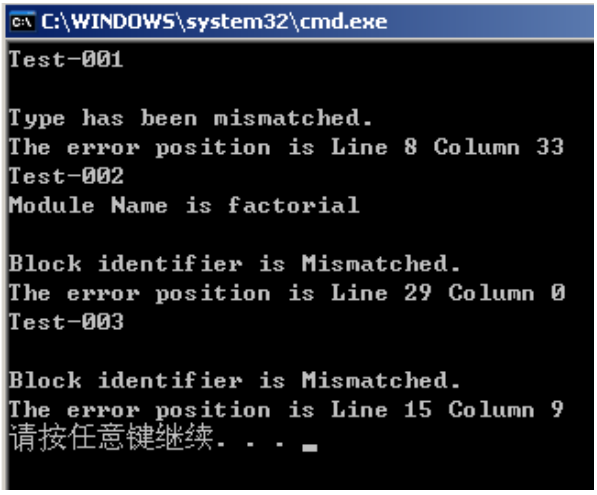


- 测试词法错误变异程序（测试源程序见 testcases）：

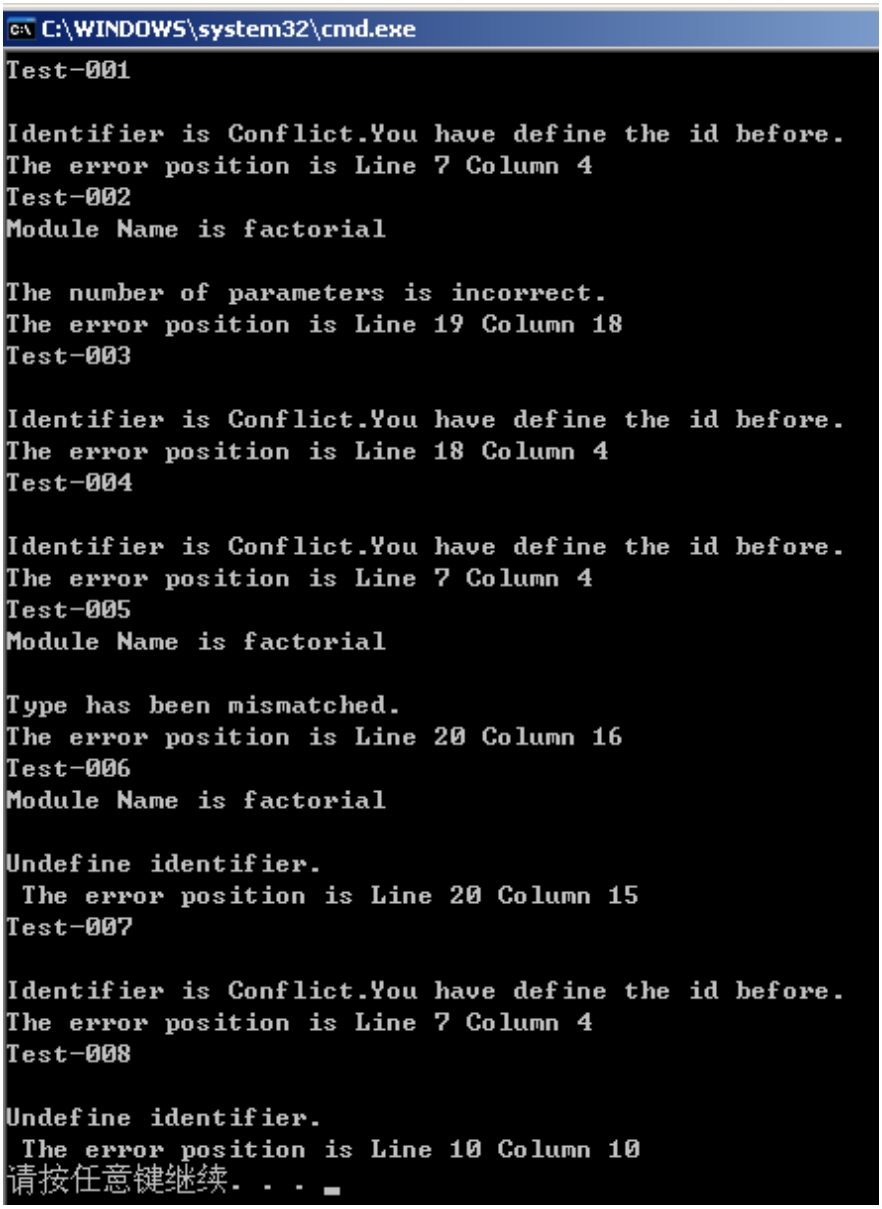




- 测试语法错误变异程序（测试源程序见 testcases）：



- 测试语义错误变异程序（测试源程序见 testcases）：



五、 实验心得

这次实验确实比较难，JavaCUP 文件差不多 1100 多行代码，后期在修改的时候找一条文法都比较麻烦，生怕修改一个地方导致所有的代码都面目全非。但是总算是基本完成了，实现了部分异常处理以及函数调用图的绘制。通过这次实验，不但熟悉了 JavaCUP 的文件结构，还有 Java 本身，还在调试程序中以及对生成文件的学习中对编译原理中 LALR 语法分析的过程以及语法制导翻译有了较深的理解，对学习编译原理提供了很大的帮助。

虽然花费了很长时间，实现了这个功能有待完善的半成品，还是成就感满满啊~

Thanks\(^o^)/~