

编译原理实验

复杂性度量工具 ALIOTH

目 录

1	简介	3
1.1	实验目的.....	3
1.2	实验环境.....	4
1.2.1	编程语言	4
1.2.2	开发工具.....	4
1.2.3	设计风格.....	4
1.2.4	编码规范.....	4
1.3	实验计划.....	4
1.4	诚信声明.....	4
2	实验内容	6
2.1	实验一、熟悉 Oberon-0 语言定义	6
2.1.1	实验步骤 1.1、编写一个正确的 Oberon-0 源程序.....	6
2.1.2	实验步骤 1.2、编写上述 Oberon-0 源程序的变异程序.....	6
2.1.3	实验步骤 1.3、讨论 Oberon-0 语言的特点.....	7
2.1.4	实验步骤 1.4、讨论 Oberon-0 文法定义的二义性.....	7
2.1.5	提交结果.....	7
2.1.6	评价标准.....	7
2.2	实验二、自动生成词法分析程序 (JFlex)	8
2.2.1	实验步骤 2.1、总结 Oberon-0 语言的词汇表.....	8
2.2.2	实验步骤 2.2、抽取 Oberon-0 语言的词法规则.....	8
2.2.3	实验步骤 2.3、下载词法分析程序自动生成工具 JFlex	8
2.2.4	实验步骤 2.4、生成 Oberon-0 语言的词法分析程序.....	9
2.2.5	实验步骤 2.5、讨论不同词法分析程序生成工具的差异	9
2.2.6	提交结果.....	9
2.2.7	评价标准.....	10
2.3	实验三、自动生成语法分析程序 (JavaCUP)	10
2.3.1	实验步骤 3.1、下载自动生成工具 JavaCUP	10
2.3.2	实验步骤 3.2、配置和试用 JavaCUP	10
2.3.3	实验步骤 3.3、生成 Oberon-0 语法分析和语法制导翻译程序.....	11
2.3.4	实验步骤 3.4、讨论不同生成工具的差异	11
2.3.5	提交结果.....	11
2.3.6	评价标准.....	12
2.4	实验四、手工编写递归下降预测分析程序.....	12
2.4.1	实验步骤 4.1、设计 Oberon-0 语言的翻译模式.....	12
2.4.2	实验步骤 4.2、编写递归下降预测分析程序.....	12

2.4.3	实验步骤 4.3、语法分析讨论：自顶向下 vs. 自底向上	13
2.4.4	提交结果.....	13
2.4.5	评价标准.....	13
3	Oberon-0 语言	15
3.1	简介	15
3.2	词法定义.....	16
3.2.1	单词.....	16
3.2.2	基本数据类型.....	16
3.3	语法定义.....	16
3.3.1	扩展 BNF (EBNF)	16
3.3.2	Oberon-0 语言的 EBNF 定义	16
3.4	语言描述.....	18
3.4.1	模块声明.....	18
3.4.2	运算符与表达式.....	18
3.4.3	类型声明.....	18
3.4.4	选择符.....	19
3.4.5	作用域规则.....	19
3.4.6	过程声明与参数传递.....	19
3.4.7	预定义函数.....	19
3.5	例子程序.....	20
4	复杂性度量模型.....	22
4.1	简介	22
4.2	计算模型.....	22
4.3	计算方法.....	23
4.3.1	数据类型的复杂度.....	23
4.3.2	常量声明的复杂度.....	24
4.3.3	类型声明的复杂度.....	24
4.3.4	变量声明的复杂度.....	24
4.3.5	数据复杂度计算示例.....	25
4.3.6	表达式的复杂度.....	26
4.3.7	语句的复杂度.....	26
4.3.8	语句复杂度计算示例.....	26
4.3.9	其他部分复杂度.....	28
4.3.10	其他部分复杂度计算示例.....	28
5	实验软装置	29
5.1	目录和文件组织.....	29
5.2	异常类设计.....	29
5.2.1	异常类程序包.....	29
5.2.2	通用异常类.....	29
5.3	词法分析实验软装置.....	30
5.3.1	词法错误的异常类.....	30
5.3.2	词法分析主程序.....	31
5.3.3	词法分析程序测试用例.....	31
5.4	语法分析和语法制导翻译实验软装置.....	31
5.4.1	语法错误的异常类.....	32
5.4.2	语义错误的异常类.....	32
5.5	自动化测试 API	32

1 简介

在开始实验之前，请务必花足够时间阅读完本文档关于实验的描述与约定！

1.1 实验目的



本实验是编译原理实验环节的一个综合型、应用型实验，其中还包含部分研究与探索型实验内容。

本实验处理的程序设计语言是 Oberon-0，它是著名的 Pascal 和 Modula-2 语言的后继者 Oberon 语言的一个精简子集。学生在本实验中需开发一个面向 Oberon-0 的源程序复杂性度量工具，该工具基于一个语法制导的（Syntax-Directed）源程序复杂性计算模型，根据一个输入的 Oberon-0 源程序自动计算出该源程序中代码的复杂度（度量源程序复杂性的一个整数值）。本实验项目将该工具命名为 ALIOTH，其中文名为“玉衡”，乃北斗七星之一。

本实验借助于 ALIOTH 工具的设计与实现，帮助学生通过实践深入理解和牢固掌握编译技术中的词法分析、语法分析、语法制导翻译、自动生成工具等重要环节。

本实验的主要目标包括：

- 掌握词法分析程序的工作原理与构造方法，并能够推广到对文本的串匹配搜索等其他同类型应用。
- 掌握词法分析程序自动生成工具 lex 或类似工具的工作原理与使用方法，学习如何编写一个 lex 源文件以解决词法分析或模式匹配问题，在实践中进一步体会软件自动化的基本思路。
- 掌握递归下降的预测分析方法以及语法制导的翻译技术，学习如何根据 BNF 语法定义和应用需求设计一个翻译模式（Translation Scheme），并利用高级程序设计语言的递归机制实现一个翻译模式。
- 掌握语法分析程序自动生成工具 yacc 或类似工具的工作原理与使用方法，学习如何编写一个 yacc 源文件以解决语法分析及语法制导翻译问题，进一步加深体会软件自动化的基本思路。
- 通过加强设计空间（Design Sapce）的深入探讨以及 Java 编程风格的实践，提高对面向对象设计的认识，养成良好的编程习惯与规范，并学会多个工程文档的组织与交付。

1.2 实验环境

软件工具 ALIOTH 基于当前主流的面向对象开发平台，编码风格遵循主流的参考规范。

1.2.1 编程语言

本实验要求采用 Java 语言完成，开发平台使用 JDK 1.5+版本。

1.2.2 开发工具

学生可自由选择 Eclipse、JBuilder 等 IDE 环境，也可直接采用 UltraEdit、EditPlus 等编辑器在命令行工作。但提交的实验结果必须独立于特定的 IDE，可直接运行在 JDK 上。

1.2.3 设计风格

本实验项目要求基于面向对象风格来设计 ALIOTH 的所有类，并遵循一种良好的程序设计习惯。

例如，如果你将一个程序的所有功能全部放在一个硕大的 `main()` 方法中实现，这样的设计与编码风格会被扣分。

1.2.4 编码规范

学生在 ALIOTH 实验过程中应注意培养规范的编码风格。本实验要求所有源代码严格遵循 Sun 公司（现为 Oracle 公司）关于 Java 程序设计语言的编码规范（Code Conventions for the Java Programming Language, Revised April 1999），参见：

<https://www.oracle.com/java/technologies/javase/codeconventions-contents.html>。

完成实验后，应使用 JDK 附带的文档工具 `javadoc`，根据源程序中的文档化注释自动生成相应的 HTML 格式的设计文档。

1.3 实验计划

ALIOTH 实验项目是编译原理课程的一个综合型、应用型实验，为高质量地完成这一课程实践环节，你需要在开展实验之前精心规划自己的时间与精力投入，因而推荐你参考以下建议及时开展实验活动。

根据任课教师对往届学生完成编译原理类似实验项目的经验估计，你在语言预备实验（实验一）的投入时间不足 1 天；在词法分析阶段的实验（实验二）约为 3 天；在语法分析阶段及语法制导翻译阶段的实验（实验三和实验四）约为 10 天。另，不要忘记撰写、整理和提交实验文档还需要一定的时间！

1.4 诚信声明

你必须独立完成编译原理课程实践环节的所有实验步骤，并独立撰写实验报告和编写、调试程序。在实验过程中，你可以参考各种各样的资源，并与其他同学讨论问题（并且鼓励在实验过程中与其他同学讨论问题），但复制其他同学实验结果中的任何代码片段或任何文档片段都会被视作抄袭行为。

一旦发现有任何抄袭行为，你在本课程实践环节的所有成绩将被取消，同时也意味着你本门课程的不及格。情节严重者，整门课程的总评成绩将作为“作弊 0 分”呈报，这也将导致你丧失申请中山大学学士学位的机会。

望诸位同学慎之、戒之。切勿心存侥幸，切勿挑战老师的智慧！

2 实验内容

对于一个像 **ALIOTH** 这样大规模、复杂实验项目而言，精心分解实验的过程对实验的执行效果会产生重要影响。本实验分解为 4 个子实验项目，每一子实验项目都规定了明确的实验步骤、实验要求以及需要提交的实验结果。

2.1 实验一、熟悉 Oberon-0 语言定义

本文档第 3 部分给出了 Oberon-0 语言的完整 BNF 定义，即 Oberon-0 语言在语法方面的形式化规格说明（Specification）；此外，第 3 部分还以自然语言的方式提供了 Oberon-0 语言的语义描述。

请仔细阅读 Oberon-0 语言的定义，并完成以下实验内容。

2.1.1 实验步骤 1.1、编写一个正确的 Oberon-0 源程序

遵循 Oberon-0 语言的 BNF 定义，编写一个正确的 Oberon-0 源程序。要求在这个源程序中，用到 Oberon-0 语言的所有语法构造，即你编写的源程序覆盖了 Oberon-0 语言提供的模块、声明（类型、常量、变量等）、过程声明与调用、语句、表达式等各种构造。

如果有可能，你编写的 Oberon-0 源程序最好是有其实际意义的，譬如一个求阶乘的程序、一个求最大公因子的程序、一个用加法实现乘法的程序等等，但这只是一个任选的要求。

注意，这里仅要求你编写一个词法、语法和语义符合 Oberon-0 语言定义的源程序，并未强制要求该源程序在逻辑上是正确的。

2.1.2 实验步骤 1.2、编写上述 Oberon-0 源程序的变异程序

根据上述正确的源程序，写出若干含有词法、语法或语义错误的 Oberon-0 源程序。其做法是对原有的正确源程序做出最小的改动，使之成为一个包含至少一个错误的源程序；为每一种可能存在的词法、语法和语义错误构造出一个变异程序。例如，有一个变异程序中含有不合法的标识符，有另一个变异程序中含有不合法的常量，还有一个变异程序中丢失了左括号等。在每一变异程序的第一行用注释说明该变异产生的错误类型。

基于极限编程 XP（eXtreme Programming）中的测试驱动编程（Test-Driven Programming）思想，在进入正式编码前先设计好测试用例将有助于把握正确的设计。而上述正确的 Oberon-0 源程序及其一批含有错误的变异程序将在后述实验步骤中作为测试用例。

上述“程序变异”的思路源于软件测试技术中的变异测试（Mutation Testing）。变异测试的目标是筛选

更有效的测试数据，其思路是通过变异（Mutation）操作在程序中植入一个错误，变异后的程序称为变异程序（Mutant）；通过检查测试数据能否发现变异程序中的这些错误，可判断测试数据的有效性。有兴趣的同学不妨查阅相关资料（例如：http://en.wikipedia.org/wiki/Mutation_testing）进一步学习变异测试技术。

2.1.3 实验步骤 1.3、讨论 Oberon-0 语言的特点

在 Oberon-0 语言中，与其前辈 Pascal 语言一样区别了保留字（Reserved Word）与关键字（Keyword）两个概念。例如，**IF**、**THEN**、**ELSIF** 等是保留字，而 **INTEGER**、**WRITE**、**WRITELN** 则是关键字。试解释这两个概念的区别。

根据 Oberon-0 语言的 BNF 定义，Oberon-0 程序中的表达式语法规则与 Java、C/C++ 等常见语言的表达式有何不同之处？试简要写出它们的差别。

2.1.4 实验步骤 1.4、讨论 Oberon-0 文法定义的二义性

根据 Oberon-0 语言的 BNF 定义，讨论 Oberon-0 程序的二义性问题，即讨论根据上述 BNF 定义的上下文无关文法是否存在二义性。

如果你认为该文法存在二义性，则举例说明在什么地方会出现二义性，并探讨如何改造文法以消除二义性。

如果你认为该文法没有二义性，则请解释：为何在其他高级程序设计语言中常见的那些二义性问题的 Oberon-0 语言中并未出现？

2.1.5 提交结果

将实验一的所有结果存放在子目录 **xxxxxxxxNNN\ex1** 中，其中 **xxxxxxxx** 是你的学号，**NNN** 是你的中文姓名。例如，学号为“05372001”、姓名为“马婷”的同学，应将其完成的实验一全部结果存放在“**05372001 马婷\ex1**”子目录中；注意学号与姓名之间不要加空格、姓与名之间亦不要加空格。

实验一最终提交的实验结果应包括：

- 在 **ex1** 子目录中存放自述文件 **readme.txt**，其中给出你的姓名、学号、电子邮件、联系电话、完成日期、以及其他补充说明。
- 在 **ex1** 子目录中存放你的实验报告 **Oberon-0.pdf**（允许、但不建议采用 **.doc**、**.docx**、**.txt**、**.wps** 等格式；下同），其中包括 Oberon-0 语言特点介绍、文法二义性讨论、实验心得体会等内容。
- 在 **ex1\testcases** 子目录中存放一个词法、语法、语义均正确的 Oberon-0 程序源代码，该文本文件的文件名由你自己根据程序的功能自由命名，但必须以 **.obr** 为文件扩展名。
- 在 **ex1\testcases** 子目录中存放一批含有词法、语法或语义错误的变异程序，文件名均同语法正确的 Oberon-0 源程序，但分别以 **.xxx** 为文件扩展名（**xxx** 表示序号，从 **001** 开始）。

2.1.6 评价标准

为了更好地分配时间与精力完成实验一，你可参考以下评分标准：

项 目	权重（%）
1、一个正确的 Oberon-0 源程序	20

2、一批含有词法、语法或语义错误的变异程序	40
3、Oberon-0 语言特点的讨论	10
4、Oberon-0 文法二义性的讨论	20
5、文档组织及 readme 文件等	10

2.2 实验二、自动生成词法分析程序 (JFlex)

实验二要求你下载一个词法分析程序自动生成工具 JFlex，并利用该工具自动产生 Oberon-0 语言的词法分析程序，该词法分析程序的源代码是用 Java 语言编写的。

2.2.1 实验步骤 2.1、总结 Oberon-0 语言的词汇表

根据 Oberon-0 语言的 BNF 定义和语言描述，抽取 Oberon-0 语言的词汇表以供词法分析程序设计与实现之用。你需要将 Oberon-0 的所有单词分类，并以表格形式列出各类词汇的预定义单词；譬如，在保留字表中列出所有的保留字，在运算符表中列出所有的运算符，等等。

请在实验报告中说明你的单词分类的理由，并解释如何处理 Oberon-0 语言中保留字和关键字两类略有不同的单词。

2.2.2 实验步骤 2.2、抽取 Oberon-0 语言的词法规则

在 Oberon-0 语言的 BNF 定义中，既包括 Oberon-0 语言的语法定义部分，也包括 Oberon-0 语言的词法定义部分。请将词法定义从 Oberon-0 语言的 BNF 中分离出来，并写成正则定义式 (Regular Expression) 的形式。

然后，在实验报告中讨论与 Pascal、C/C++、Java 等常见高级程序设计语言的词法规则相比，Oberon-0 语言的词法规则有何异同。

2.2.3 实验步骤 2.3、下载词法分析程序自动生成工具 JFlex

实验二指定的词法分析程序自动生成工具选用由 Gerwin Klein 开发的 JFlex。这是一个类似 Unix 平台上 lex 程序的开源 (Open Source) 软件工具，遵循 GNU General Public License (GPL)。JFlex 本身采用 Java 语言编写，并且生成 Java 语言的词法分析程序源代码。该软件工具的前身是由美国普林斯顿大学计算机科学系 Elliot Berk 开发、C. Scott Ananian 负责维护的 JLex。


从 <http://www.jflex.de/> 可下载该工具，当前的最新稳定版本是 2020 年 5 月发布的 JFlex 1.8.2。该网站提供的压缩文件中已包含了你在本实验中所需的各类资源，包括该工具的 Java 源代码、支持运行的库文件与脚本文件、用户文档、输入源文件例子等。

根据你自己的安装配置，修改 JFlex 安装目录下脚本文件 `bin\jflex.bat` 中的两个环境变量 `JFLEX_HOME` 和 `JAVA_HOME` 的设置。然后运行 JFlex 附带的输入源文件例子，以验证你是否正确安装并配置了 JFlex。

如果你觉得 JFlex 附带的用户手册仍不足以帮助你掌握 JFlex 的原理或用法，自己动手在网上查找其他关于 JLex、GNU Flex、lex 等类似工具的大量电子资源。

2.2.4 实验步骤 2.4、生成 Oberon-0 语言的词法分析程序

仔细阅读 JFlex 的使用手册，根据你在实验步骤 2.1 给出的 Oberon-0 语言词法规则的正则定义式，编写一个 JFlex 输入源文件。

本实验提供的实验软装置为实验二预定义了代表 Oberon-0 语言各种词法错误的异常类，并为实验二设计了主程序 `main()`。你在实验二中不要改变 `main()` 的内容，并且在你的词法分析程序中应根据词法错误的类别抛出相应的异常类。实验软装置还提供了一批测试用例，用于检测你的程序是否能够识别出词法错误并正确地对词法错误进行分类（通过抛出的异常类标识错误类别）。关于  实验软装置的详细描述请参阅本文档第 5 部分。

以你编写的源文件作为输入运行 JFlex，得到一个 Oberon-0 语言词法分析程序的 Java 源代码；编译该源程序生成 Java 字节码。通过实验软装置提供的测试用例检查后，利用你在实验一编写的所有 Oberon-0 源程序测试你的词法分析程序。

2.2.5 实验步骤 2.5、讨论不同词法分析程序生成工具的差异

比较以下 3 种流行的词法分析程序自动生成工具之间的差异：JFlex、JLex 和 GNU Flex。主要讨论这些软件工具接收输入源文件时，在词法规则定义方面存在的差异。

在网站 <http://www.cs.princeton.edu/~appel/modern/java/JLex/> 可找到关于 JLex 工具的权威资料；关于 GNU Flex 的官方资料则位于 <https://github.com/westes/flex>。

2.2.6 提交结果

将实验二的所有结果存放在子目录 `xxxxxxxxNNN\ex2` 中，其中 `xxxxxxxx` 是你的学号，`NNN` 是你的中文姓名，命名规则同实验一。

实验二最终提交的实验结果应包括：

- 在 `ex2` 子目录中存放自述文件 `readme.txt`，其中给出你的姓名、学号、电子邮件、联系电话、完成日期、以及其他补充说明。
- 在 `ex2` 子目录中存放你在实验二撰写的实验报告 `lexgen.pdf`，其中的内容包括：
 - 以表格形式列出 Oberon-0 语言的词汇表。
 - 用正则定义式描述 Oberon-0 语言词法规则；若你使用纯文本书写正则定义式，其中的元符号“定义为”使用“->”表示，空串使用“epsilon”表示。
 - 紧随 Oberon-0 语言词法规则之后，给出一段关于 Oberon-0 语言与其他高级语言的词法规则之异同比较。
 - 讨论 3 种不同 lex 族软件工具的输入文件中，词法规则定义的差异或特点。
- 自动生成的 Oberon-0 语言词法分析程序及其测试用例，包括：
 - 在 `ex2` 子目录中存放各种脚本文件，包括：运行 JFlex 根据输入文件生成词法分析程序的脚本 `gen.bat`；编译词法分析程序的脚本 `build.bat`；运行词法分析程序扫描你编写的正确 Oberon-0 例子程序的脚本 `run.bat`；运行所有由实验软装置提供的测试用例的脚本 `test.bat`。

- 在 **ex2\src** 子目录中存放面向 Oberon-0 语言的 JFlex 输入源文件 **oberon.flex** 以及你所需的其他相应文件。
- 在 **ex2\src** 子目录中存放由 JFlex 生成的 Oberon-0 语言词法分析程序的 Java 源程序 **OberonScanner.java**（注意我们不使用默认的生成名字 **Yylex**）。
- 在 **ex2\bin** 子目录中存放根据 Oberon-0 词法分析程序源代码编译得到的字节码文件 **OberonScanner.class** 以及其他相关的字节码文件。
- 在 **ex2\doc** 子目录中存放根据你的源程序中文档化注释自动生成的 javadoc 文档。
- 在 **ex2\jflex** 子目录中存放你所使用的 JFlex 工具。

注意，请不要提交那些由 IDE 自动生成的配置文件或备份文件（下同）。

2.2.7 评价标准

为了更好地分配时间与精力完成本次实验，你可参考以下评分标准：

项 目	权重（%）
1、Oberon-0 语言词法规则的正则定义式	20
2、JFlex 输入源文件的正确性	40
3、生成的词法分析程序的运行与测试	20
4、不同 lex 族工具差异比较	10
5、文档组织及 readme 文件等	10

2.3 实验三、自动生成语法分析程序 (JavaCUP)

实验三要求你下载一个语法分析程序自动生成工具 JavaCUP，利用该工具自动产生一个 Oberon-0 语言的语法分析和语法制导翻译程序；生成的程序源代码是以 Java 语言编写的。

2.3.1 实验步骤 3.1、下载自动生成工具 JavaCUP

实验三选用最早由美国卡内基·梅隆大学的 Scott E. Hudson 开发的一个语法分析程序自动生成工具 JavaCUP，它是一个 LALR Parser Generator。JavaCUP 是一个类似 Unix 平台上 yacc 程序的开源（Open Source）软件工具，遵循 GNU General Public License（GPL）。JavaCUP 本身采用 Java 编写，并且生成 Java 语言的分析程序源代码。该软件工具经由美国普林斯顿大学计算机科学系 Andrew W. Appel 教授指导 Frank Flannery 等人改进，目前由 C. Scott Ananian 负责维护。

从 <http://www2.cs.tum.edu/projects/cup/> 可下载该软件工具的最新版本 CUP 0.11b。该网站已包含了你在实验中所需的各类资源，包括该工具的 Java 源代码、已编译生成的字节码、简明的用户手册、以及一个简单的命令行计算器例子等。

2.3.2 实验步骤 3.2、配置和试用 JavaCUP

成功下载并配置后，试运行 JavaCUP 附带的输入源文件例子（一个基于命令行的简单计算器应用），以保证你正确安装并配置了 JavaCUP。

如果你觉得 JavaCUP 附带的用户手册仍不足以帮助你掌握 JavaCUP 的原理与用法，自己动手在网上查找其他关于 GNU Bison、yacc 等类似工具的大量电子资源。

2.3.3 实验步骤 3.3、生成 Oberon-0 语法分析和语法制导翻译程序

仔细阅读 JavaCUP 使用手册，根据 Oberon-0 语言的 BNF 定义编写一个 JavaCUP 输入源文件。

根据你的 JavaCUP 输入源文件生成的语法分析程序须完成以下功能：

1、对于一个存在词法、语法或语义错误的 Oberon-0 源程序，必须至少指出一处错误，并判断错误的类别及产生错误的位置（错误产生的位置定位允许有偏差），并以相应的异常对象向客户程序报告找出的错误的类别。是否支持其他功能取决于你的时间、精力与能力，譬如你可尝试从错误中恢复并继续执行语法分析，也可生成设计图时立即中止程序的执行。在错误检查与错误恢复（指找到错误后继续执行分析过程的能力）方面做得优秀的实验可获得更高的评分。

2、对于一个词法、语法和语义完全正确的 Oberon-0 源程序，自动计算出该源程序的复杂度。本文档第 4 部分详细介绍了 **ALIOTH** 工具所基于的语法制导复杂性度量模型。

你在生成 Oberon-0 语言的语法分析程序时，可以直接使用实验二由 JFlex 生成的 Oberon-0 语言的词法分析程序。

以你编写的源文件为输入运行 JavaCUP，得到 Oberon-0 语言语法分析程序的若干 Java 源代码；编译这些 Java 源程序生成相应的字节码，再分别利用自己在实验一编写的 Oberon-0 源程序及其变异程序测试生成的结果，看看你生成的语法分析和语法制导翻译程序能否正确地识别出 Oberon-0 源程序中的各类错误，并且能否针对正确的 Oberon-0 源程序输出其源程序复杂度，并通过实验软设置给出的自动化测试工具验证正确性。

2.3.4 实验步骤 3.4、讨论不同生成工具的差异

比较两种流行的语法分析程序自动生成工具之间的差异：JavaCUP 和 GNU Bison，主要讨论这两种软件工具接收输入源文件时，在语法规则定义方面存在的差异。关于 GNU Bison 工具的官方资料可在网站 <http://www.gnu.org/software/bison/bison.html> 找到。

同样基于 Java 语言的分析器生成工具（Parser Generator，即 Compiler Compiler），还有一个名为 JavaCC 的工具。在网上搜索并浏览关于 JavaCC 的相关信息，用最扼要的一两句话指出 JavaCC 与 JavaCUP 的最核心区别。

2.3.5 提交结果

将实验三的所有结果存放在子目录 **xxxxxxxxNNN\ex3** 中，其中 **xxxxxxxx** 是你的学号，**NNN** 是你的中文姓名，命名规则同实验一。

实验三最终提交的实验结果应包括：

- 在 **ex3** 子目录中存放自述文件 **readme.txt**，其中给出你的姓名、学号、电子邮件、联系电话、完成日期、以及其他补充说明。
- 在 **ex3** 子目录中存放讨论两个不同 yacc 族工具语法规则定义差异的文档 **yaccgen.pdf**。
- 自动生成的 Oberon-0 语言语法分析程序及其测试用例，包括：
 - 在 **ex3** 子目录中存放各种脚本文件，包括：运行 JavaCUP 根据输入文件生成语法分析程序的脚本 **gen.bat**；编译语法分析程序的脚本 **build.bat**；运行语法分析程序处理你编写的正确 Oberon-0 例子程序的脚本 **run.bat**。

- 在 **ex3\src** 子目录中存放面向 Oberon-0 语言的 JavaCUP 输入文件 **oberon.cup** 以及你所需的其他相应文件。
- 在 **ex3\src** 子目录中存放由 JavaCUP 生成的 Oberon-0 语法分析程序源代码 **Parser.java**、**Symbol.java** 和其他相应的 Java 源代码（注意我们不使用默认的生成名字 **parser** 和 **sym** 等）。
- 在 **ex3\bin** 子目录中存放根据 Oberon-0 语法分析程序源代码编译得到的字节码文件 **Parser.class** 以及其他相关的字节码文件。
- 在 **ex3\doc** 子目录中存放根据你的源程序中文档化注释自动生成的 javadoc 文档。
- 在 **ex3\javacup** 子目录中存放你所使用的 JavaCUP 工具。

2.3.6 评价标准

为了更好地分配时间与精力完成本次实验，你可参考以下评分标准：

项 目	权重（%）
1、JavaCUP 输入源文件的正确性	50
2、生成的语法分析程序的运行与测试	30
3、两种 yacc 族工具差异比较	10
4、文档组织及 readme 文件等	10

2.4 实验四、手工编写递归下降预测分析程序

实验四要求你利用 Java 语言手工编写一个 Oberon-0 语言的语法分析程序，该语法分析程序执行与实验三自动生成的语法分析程序完全相同的功能，即你需要实现 Oberon 源码复杂性度量功能，根据给出的 Oberon 程序源码计算其复杂度。

该语法分析程序采用递归下降预测分析技术，要求你遵循语法制导翻译思想，先设计 Oberon-0 语言的翻译模式，再据此编写语法分析程序。

2.4.1 实验步骤 4.1、设计 Oberon-0 语言的翻译模式

根据 Oberon-0 语言 BNF 定义中的语法规则，以及你完成 Oberon-0 源程序处理的应用需求，为 Oberon-0 语言设计一个合适的翻译模式。

注意，由于实验四规定使用递归下降的预测分析技术，你需要改造文法以适用于这种自顶向下的分析方法。例如，上下文无关文法中的左递归必须消除，以避免递归下降的预测分析程序进入死循环。

2.4.2 实验步骤 4.2、编写递归下降预测分析程序

根据上一步骤获得的翻译模式，利用 Java 语言设计并实现一个 Oberon-0 语言的递归下降预测分析程序。

结合编译原理理论课所学知识，从一个翻译模式设计一个递归下降预测分析程序已有比较成熟的启发式规则，你应遵循这些规则设计你的语法分析程序。

例如：文法的每一非终结符号应对应着一个递归子程序，开始符号则对应着其中的主程序；由向前看符号（Lookahead）决定分支动作；每一个继承属性对应一个形式参数，所有综合属性对应返回值，子结

点的每一属性对应一个局部变量；翻译模式中产生式右部的终结符号、非终结符号与语义动作分别执行匹配、递归调用和嵌入代码等动作。

2.4.3 实验步骤 4.3、语法分析讨论：自顶向下 vs. 自底向上

通过你自己在实验三和实验四的实际体会，对递归下降预测分析技术和自底向上的 LR 分析技术这两种不同的分析策略进行比较。

建议你在比较两种技术的各自优点和不足时，考虑（但不必局限于）以下方面：

- 分析技术的简单性，包括分析程序是否易于调试。
- 分析技术的通用性，即能处理的语言范围。
- 是否便于表达语义动作以完成语法制导翻译。
- 是否易于实现出错恢复。
- 若以表格驱动方式取代递归程序实现，则分析表大小的优劣如何？
- 分析速度。

2.4.4 提交结果

将实验四的所有结果存放在子目录 `xxxxxxxxNNN\ex4` 中，其中 `xxxxxxxx` 是你的学号，`NNN` 是你的中文姓名，命名规则同实验一。

实验四最终提交的实验结果应包括：

- 在 `ex4` 子目录中存放自述文件 `readme.txt`，其中给出你的姓名、学号、电子邮件、联系电话、完成日期、以及其他补充说明。
- 在 `ex4` 子目录中存放描述 Oberon-0 语言翻译模式的文本文件 `scheme.pdf`。
- 你手工编写的 Oberon-0 语言语法分析和语法制导翻译程序及其测试用例，包括：
 - 在 `ex4` 子目录中存放各种脚本文件，包括：编译语法分析程序的脚本 `build.bat`；运行语法分析程序扫描你编写的正确 Oberon-0 例子程序的脚本 `run.bat`。
 - 在 `ex4\src` 子目录中存放 Oberon-0 语法分析程序的主程序 `OberonParser.java`，以及你在考虑面向对象设计时引入的其他 Java 类的源代码；你用程序包组织的各相关类亦存放在 `ex4\src` 之中的相应子目录中。
 - 在 `ex4\bin` 子目录中存放根据词法分析程序源代码编译得到的 Java 字节码文件 `OberonParser.class` 以及其它相关的字节码文件。
 - 在 `ex4\doc` 子目录中存放根据你的源程序中文档化注释自动生成的 javadoc 文档。

2.4.5 评价标准

为了更好地分配时间与精力完成本次实验，你可参考以下评分标准：

项 目	权重 (%)
-----	--------

1、Oberon-0 语言的翻译模式	30
2、语法分析和语法制导翻译程序的源代码	40
3、语法分析和语法制导翻译程序的运行与测试	15
4、面向对象设计与编码风格	10
5、文档组织及 readme 文件等	5

3 Oberon-0 语言

本实验的处理对象是 Oberon-0 语言，该语言中包含了高级程序设计语言的表达式，以及结构化程序设计中的结构化控制结构、子程序、参数传递等机制的抽象。

3.1 简介

用于编译原理实验的计算机语言应足够简单，但又不失其代表性。据此，本实验项目选择了 Oberon-0 语言为处理对象。

Oberon-0 的来历可追溯到近 50 年前，在此期间的程序设计语言发展约每 10 年就有一标志性成果。1960 年，P. Naur 等人设计了 Algol 60 语言；约 10 年后，随着结构化程序设计思想的成熟，N. Wirth 设计出远比 Algol 68 语言成功（特别是在教育界）的 Pascal 语言；又一个 10 年过去，N. Wirth 根据程序设计和软件工程技术的最新进展，在 Pascal 基础上设计了 Modula-2 语言；又过了约 10 年后，M. Reiser 与 N. Wirth 一起，将 Pascal 语言和 Modula-2 语言的程序设计本质精华浓缩为 Oberon 语言。

Oberon-0 语言是 Oberon 语言的一个子集，为程序员提供了良好的程序结构。在 Oberon-0 程序中，最基本的语句是赋值语句；复合语句支持顺序、条件（**if** 语句）和迭代（**while** 语句）执行。Oberon-0 中还支持子程序这一重要概念，包括过程声明和过程调用两个范畴，并且提供了两种不同的参数传递方式：按值调用（值参数）和按引用调用（可变参数）。

然而 Oberon-0 的类型系统却十分简洁，仅有的基本数据类型是整数类型（**INTEGER**）和布尔类型（**BOOLEAN**），因而可声明整数类型的常量和变量，也允许用算术运算符构造表达式；而表达式的比较运算则产生 **BOOLEAN** 类型的值，并可用于逻辑运算。Oberon-0 的复合数据类型包括数组和记录，且允许任意嵌套；但最基本的指针类型或引用类型就被省略了。

一个过程代表了由语句组成的功能单元，因而在一个过程的写法中自然会关系到名字的局部性（Locality）问题。Oberon-0 语言支持将标识符声明为局部于某一过程，即仅在该过程本身范围内标识符才是可见的或合法的。

由 N. Wirth 本人编著的 *Theory and Techniques of Compiler Construction: An Introduction*（Addison-Wesley, 1996, ISBN 0-201-40353-6，本课程教学网站提供了该教材的电子版）中，第 6 章描述了 Oberon-0 程序设计语言的语法并给出一个样板程序。本文档的语法定义和例子程序即源于该教材，但作了少数改动，因而本实验处理的 Oberon-0 语言应以本文档的定义为准。

3.2 词法定义

Oberon-0 语言定义了非常简单的语法规则。

3.2.1 单词

与我们熟悉的 C、C++、Java 等语言不同，Oberon-0 语言是大小写无关的。譬如，保留字 **WHILE**、**While** 和 **while** 三种写法是等价的；而标识符 **BALANCE**、**Balance** 和 **balance** 是相同的标识符。

Oberon-0 的标识符长度不允许超过 24 个字符（允许 24 个字符）。

在 Oberon-0 中还支持括号风格的注释，在 “(” 和 “)” 之间的内容全部为注释；注意，Oberon-0 注释不允许嵌套。

3.2.2 基本数据类型

在 Oberon-0 程序中仅支持 **INTEGER** 和 **BOOLEAN** 两种基本数据类型，可以利用 **VAR** 声明这两种类型的变量。

INTEGER 类型的常量书写形式只允许 Pascal 语言的无符号整数；这些常量可以由 0 开头，但解释为八进制常量（此时常量中不允许出现 8 和 9 两个数字），非 0 开头则解释为十进制常量。无论十进制还是八进制整数常量，每一常量中包含数字（包括 0）的个数不可超过 12 个（从而限制了整数常量允许表达范围的最大值）。

INTEGER 类型常量与标识符之前必须以空白符号分隔；例如，扫描 **25id** 时应作为一个非法整数常量处理，而不是理解为常量 **25** 和标识符 **id** 两个单词。注意，不支持书写 **BOOLEAN** 类型的常量 **TRUE** 和 **FALSE**。

3.3 语法定义

本小节以 EBNF 定义了 Oberon-0 语言的形式语法。

3.3.1 扩展 BNF（EBNF）

EBNF 意即扩展的 BNF（Extended BNF），是我们在实际应用中定义一门计算机语言的形式语法的国际标准，参见 ISO/IEC 14977: 1996(E). *The Standard Metalanguage Extended BNF*（本课程教学网站提供了该标准的电子版文档）。

3.3.2 Oberon-0 语言的 EBNF 定义

```
module          = "MODULE" identifier ";"
                 declarations
                 ["BEGIN"
                  statement_sequence]
                 "END" identifier "." ;

declarations    = ["CONST" {identifier "=" expression ";"}]
                 ["TYPE" {identifier "=" type ";"}]
```

```
["VAR" {identifier_list ":" type ";"}]
{procedure_declaration ";"} ;

procedure_declaration = procedure_heading ";"
                      procedure_body ;

procedure_body        = declarations
                      ["BEGIN"
                       statement_sequence]
                      "END" identifier ;

procedure_heading     = "PROCEDURE" identifier [formal_parameters] ;

formal_parameters     = "(" [fp_section {";" fp_section}] ")" ;

fp_section            = ["VAR"] identifier_list ":" type ;

type                 = identifier | array_type | record_type | "INTEGER"
                      | "BOOLEAN";

record_type          = "RECORD"
                      field_list
                      {";" field_list}
                      "END" ;

field_list            = [identifier_list ":" type] ;

array_type            = "ARRAY" expression "OF" type ;

identifier_list       = identifier {"," identifier} ;

statement_sequence    = statement {";" statement} ;

statement             = [assignment |
                        procedure_call |
                        if_statement |
                        while_statement] ;

while_statement       = "WHILE" expression "DO"
                        statement_sequence
                        "END" ;

if_statement          = "IF" expression "THEN"
                        statement_sequence
                        {"ELSIF" expression "THEN"
                         statement_sequence}
                        ["ELSE"
                         statement_sequence]
                        "END" ;

procedure_call        = identifier [actual_parameters] ;

actual_parameters     = "(" [expression {"," expression}] ")" ;

assignment           = identifier selector "==" expression ;

expression            = simple_expression
                      [("=" | "#" | "<" | "<=" | ">" | ">=")
                       simple_expression] ;
```

```

simple_expression      =  ["+" | "-"] term { ("+" | "-" | "OR") term } ;
term                  =  factor { ("*" | "DIV" | "MOD" | "&") factor } ;
factor                =  identifier selector |
                        number |
                        "(" expression ")" |
                        "~" factor ;
number                =  integer ;
selector              =  { "." identifier | "[" expression "]" } ;
integer               =  digit { digit } ;
identifier             =  letter { letter | digit } ;

```

3.4 语言描述

在 *Theory and Techniques of Compiler Construction: An Introduction* 中并未详细描述 Oberon-0 语言的语义特性，因为该语言的语义可以按直观的方式参照其他程序设计语言（特别是沿 Algol 60、Pascal、Modula-2、Oberon 这一家族的语言）来理解。

本小节特别强调了 Oberon-0 可能引起理解上二义性的几个方面，使得在本实验项目中老师和所有学生对该语言的理解是无二义的。

3.4.1 模块声明

一个模块中可声明类型、常量、变量、过程等，也可声明该模块的主程序（程序体包括在 **BEGIN** 和 **END** 之间）。但一个模块也允许没有主程序，仅声明一些类型、常量、变量、过程等供其他模块使用（尽管本实验项目中尚未更深入地考虑这一问题）。

一个模块声明的 **END** 之后的标识符必须与该模块的名字相同。

3.4.2 运算符与表达式

算术表达式支持一元运算“+”和“-”，分别表示取正和取负运算；支持二元运算“+”、“-”、“*”、“**DIV**”和“**MOD**”，分别表示加法、减法、乘法、整除、取模运算。参与算术表达式的运算量必须是 **INTEGER** 类型，否则会产生类型不兼容错误；由于 Oberon-0 语言不支持实数除法运算（仅支持整除运算），因而算术表达式的运算结果总是 **INTEGER** 类型。

算术表达式可参与“=”、“#”（不等于）、“<”、“<=”、“>”和“>=”等关系运算，运算结果为 **BOOLEAN** 类型的值。**BOOLEAN** 类型的值不允许参与任何关系运算，否则会产生类型不兼容错误。

但是 **BOOLEAN** 类型的值可能参与逻辑运算“&”、“**OR**”和“~”，分别表示“与”、“或”和“非”。**INTEGER** 类型的值不允许参加任何逻辑运算，否则会产生类型不兼容错误。

3.4.3 类型声明

Oberon-0 允许类型声明，即以一个标识符重命名一个基本数据类型或复合数据类型。例如，

TYPE

```
UserId = INTEGER;  
VisitRecord = RECORD  
    user: UserId;  
    visits: INTEGER  
END;
```

此后，即可用这些标识符作为类型，以声明其他变量。例如，

VAR

```
id1, id2: UserId;  
rec: VisitRecord;
```

3.4.4 选择符

Oberon-0 语言支持两种选择符：“.”用于访问记录中的一个域；“[]”用于以下标表达式访问一个数组的元素。这两种选择符均可以嵌套使用，但必须保证：“.”左边的标识符指称的是一个记录，右边的标识符是该记录中的一个域；“[]”左边的标识符指称的是一个数组，“[]”中间是一个求值结果为 **INTEGER** 的表达式。

例如，以下选择符的使用是合法的：

VAR

```
accountList: ARRAY 100 OF RECORD  
    account: INTEGER;  
    balance: INTEGER  
END;  
BEGIN  
    accountList[1].account := 101;  
    accountList[1].balance := 8500;  
    ...
```

3.4.5 作用域规则

Oberon-0 语言是一个块（Block）结构语言；凡在一个块中声明的所有标识符，其作用域仅局限在该块之中。

例如，在一个 **MODULE** 中声明的所有类型、常量和变量（参见语法单位 declarations）在该模块中是可见的，包括该模块定义的所有块（过程声明）中都是可见的；而在一个 **PROCEDURE** 中声明的所有类型、常量和变量（参见语法单位 declarations 和 FormalParameters）则仅在该过程中是可见的，其他过程中不可见之。

3.4.6 过程声明与参数传递

如果一个过程声明的某个形式参数之前声明有保留字 **VAR**，则该参数称为一个可变参数，将采用按引用传递（Call by Reference）的参数传递方式，因而该过程可能有副作用；否则，该参数称为一个值参数，将采用按值传递（Call by Value）的参数传递方式，因而该过程不会产生副作用。严格的语义检查应保证可变参数必须是一个左值（L-value）。

一个过程声明的 **END** 之后的标识符必须与该过程的名字相同。

3.4.7 预定义函数

Oberon-0 提供了 3 种预定义函数：**read()**、**write()** 和 **writeln**，分别表示控制台输入和输出，其

含义与 Pascal 语言中的相同。

3.5 例子程序

以下提供了以 Oberon-0 书写的一个模块（Module）的源代码清单，可能会有助于你理解 Oberon-0 语言的特点。该模块中包含了几简单的过程，这些过程的名字表达了其功能。

```
(* A sample Oberon-0 source program. *)
MODULE Sample;
  PROCEDURE Multiply;
    VAR x, y, z: INTEGER;
  BEGIN
    Read(x); Read(y);
    z := 0;
    WHILE x > 0 DO
      IF x MOD 2 = 1 THEN z := z + y END;
      y := 2 * y;
      x := x DIV 2
    END ;
    Write(x); Write(y); Write(z); WriteLn
  END Multiply;

  PROCEDURE Divide;
    VAR x, y, r, q, w: INTEGER;
  BEGIN
    Read(x); Read(y);
    r := x; q := 0; w := y;
    WHILE w <= r DO
      w := 2 * w
    END;
    WHILE w > y DO
      q := 2 * q;
      w := w DIV 2;
      IF w <= r THEN
        r := r - w;
        q := q + 1
      END
    END;
    Write(x); Write(y); Write(q); Write(r); WriteLn
  END Divide;

  PROCEDURE BinSearch;
    VAR i, j, k, n, x: INTEGER;
        a: ARRAY 32 OF INTEGER;
  BEGIN
    Read(n);
    k := 0;
    WHILE k < n DO
      Read(a[k]);
      k := k + 1
    END;
    Read(x);
    i := 0; j := n;
    WHILE i < j DO
      k := (i + j) DIV 2;
      IF x < a[k] THEN
```

```
        j := k
    ELSE
        i := k + 1
    END
END;
Write(i); Write(j); Write(a[j]); WriteLn
END BinSearch;
END Sample.
```

4 复杂性度量模型

本实验项目要求开发的软件工具 ALIOTH 可根据一个输入的 Oberon-0 语言源程序，自动计算出该源程序的复杂度，即表示该源程序代码复杂性的一个整数值。本章详细描述了 ALIOTH 工具在计算复杂度所依据的计算标准，即一个语法制导的源程序复杂性度量模型。

4.1 简介

源程序复杂性度量与算法中的时间复杂度、空间复杂度有很大区别，属两个不同领域的度量方法。源程序复杂性度量属于软件系统领域，而时间复杂度和空间复杂度属于算法领域；前者是面向编程者和设计者的度量方法，而后者则是面向机器的度量方法。

随着软件系统的发展，软件系统结构变得越来越复杂，无论是开发还是维护都是一项成本高昂的工作，人们意识到必须使软件模块化，以便于开发、测试和维护。人们可以用复杂性度量对软件的复杂度和质量进行衡量，来安排工程进度，在成本、进度和性能之间寻求平衡。一个程序的源码复杂性越高，通常表明编写该程序所需要的时间花费越高，从而可使用源程序复杂性度量评判一个程序源码所要花费的时间和精力。

本实验设计了一种语法制导的 Oberon-0 源程序复杂性度量方法，可根据一个 Oberon-0 语言程序自动计算出一个模块中所有源代码的复杂度。

4.2 计算模型

一个 Oberon-0 源程序主要包括数据声明、过程定义和语句部分，过程定义中同样有数据声明和语句部分，数据声明部分和语句部分会嵌入有表达式。所以本实验的方法基于语法制导规则，针对数据、语句和表达式三个主要部分度量源码复杂性。

具体的度量标准如下表：

类型	具体类型	符号	基础值	备注
数据	整形类型	INTEGER	+1	
	布尔类型	BOOLEAN	+1	
	自定义类型	TYPEID	+2	

类型	具体类型	符号	基础值	备注
	复合类型	RECORD	+3	
	数组类型	ARRAY	+8	
	自定义声明	TYPE	+10	
	常量声明	CONST	+5	
	变量声明	VAR	+0	
表达式	选择	[]、.	+2	
	整形运算	+、-	+2	
		*, DIV、MOD	+4	
	关系运算	>、<、>=、<=、=、#	+4	
	布尔运算	OR、&、~	+6	
	小括号	()	+6	
语句	赋值	:=	+2	
	分支	IF	$+X*(N+1)$	X 为语句复杂度，N 为层数
	循环	WHILE	$+X*(2^N)$	X 为语句复杂度，N 为层数
	调用	procedure_call	+8	
其他	过程声明	PROCEDURE	+20	

4.3 计算方法

4.3.1 数据类型的复杂度

数据类型共 5 种，分别是整型、布尔类型、复合数据类型、数组类型和自定义类型（自定义类型通过

TYPE 声明得到)。整型和布尔类型复杂度为 1，自定义类型复杂度为 2，复合数据类型和数组类型需要根据其中内容具体计算。

记录类型 RECORD 的复杂度基值为 3，再累加其内容的复杂度，即为该复合数据类型的复杂度。

数组类型 ARRAY 的复杂度基值为 8，再加上其中表达式和类型的复杂度，即为该数组类型的复杂度。

参照 FLex 风格，其语法制导定义（SDD）的语义规则如下：

type : IDENTIFIER	$$$=2;$
type : "INTEGER"	$$$=1;$
type : "BOOLEAN"	$$$=1;$
type : "RECORD" field_list "END"	$$$=3+2.val;$
type : "ARRAY" expression "OF" type	$$$=8+2.val+4.val;$
field_list : field_list ";" field_one	$$$=$1.val+3.val;$
field_list : field_one	$$$=$1.val;$
field_one : identifier_list ":" type	$$$=$1.val+3.val;$

4.3.2 常量声明的复杂度

CONST 语句的复杂度基值为 5，加上变量个数和对应的表达式复杂度，即为该 CONST 声明的复杂度。

其 SDD 规则如下：

const_declare : "CONST" const_list	$$$=5+2.val;$
const_list : const_list IDENTIFIER "=" expression ";"	$$$=$1.val+1+4.val;$

4.3.3 类型声明的复杂度

TYPE 声明语句的复杂度基值为 10，加上声明内容的复杂度，即为该 TYPE 声明的复杂度。

其 SDD 规则如下：

type_declare : "TYPE" type_list	$$$=10+2.val;$
type_list : type_list IDENTIFIER "=" type ";"	$$$=$1.val+1+4.val;$

4.3.4 变量声明的复杂度

VAR 声明语句的复杂度基值为 0，其复杂度取决于声明变量的个数和变量对应的类型的复杂度。每声明

一个变量，其复杂度加 1，并且累加上 VAR 中的类型复杂度。

其 SDD 规则如下：

var_declare : VAR var_list	\$\$=\$2.val;
var_list : var_list identifier_list ":" type ";"	\$\$=\$1.val+\$2.val+\$4.val;
identifier_list : identifier_list "," IDENTIFIER	\$\$=\$1.val+1;
identifier_list : IDENTIFIER	\$\$=1;

4.3.5 数据复杂度计算示例

示例 1

```
CONST a = 1; b = 2;    (* 5 + 2 = 7 *)
```

该 CONST 语句的基本偏移量为 5，该 CONST 语句中的变量数量为 2，且两个变量对应的表达式复杂度为 0。所以这条 CONST 声明语句的复杂度为 $5 + 2 = 7$ 。

示例 2

```
TYPE
  UserId = INTEGER;
  VisitRecord = RECORD
    user: UserId;
    visits, active: INTEGER
  END;    (* 10 + 1 + 1 + 1 + 3 + 1 + 2 + 2 + 1 = 22 *)
```

TYPE 声明语句的复杂度基值为 10，语句中的变量数量为 2（TYPE 语句中每一变量的权重为 2）。第一个变量对应的数据类型为 INTEGER，其复杂度为 1；第二个变量对应的数据类型为记录类型（RECORD），其复杂度需进一步计算。记录类型的复杂度基值为 3，其中的变量个数为 2，第一个变量对应的数据类型为自定义数据类型 UserId 且变量个数为 1，故复杂度为 2；第二个变量列表对应的数据类型为 INTEGER，且变量个数为 2，故该记录数据的复杂度为 $3 + 1 + 2 + 2 + 1 = 9$ 。所以，这一条声明语句的复杂度为 $10 + 1 + 1 + 1 + 9 = 22$ 。

示例 3

```
VAR x, y, z: INTEGER;    (* 3 + 1 = 4 *)
  vs: VisitRecord;    (* 1 + 2 = 3 *)
  user: RECORD
    id, gender: INTEGER;
  END;    (* 1 + 3 + 2 + 1 = 7 *)
  arr1, arr2: ARRAY 100 OF INTEGER;    (* 2 + 8 + 1 = 11 *)
  arr3: ARRAY 100 OF RECORD
    account: INTEGER;
```

```

    balance: INTEGER
END;    (* 1 + 8 + 3 + 1 + 1 + 1 + 1 = 16 *)
(* 4 + 3 + 7 + 11 + 16 = 41 *)

```

该 VAR 声明语句包含 5 个数据声明。第一个、第二个和第三个可很直观地看出复杂度分别为 4、3 和 7。

第四个是数组类型数据，数组类型的复杂度基值为 8，声明中变量个数为 2，且数组的基类型为 INTEGER（复杂度为 1），故其复杂度为 11。

第五个同样是数组类型数据，但其数组的基类型为记录类型。数组类型的复杂度基值为 8，声明中变量个数为 1，数组基类型中的记录类型复杂度可直观看出为 7，故其复杂度为 16。

综合计算 VAR 语句的复杂度为 $4 + 3 + 7 + 11 + 16 = 41$ 。

4.3.6 表达式的复杂度

表达式的复杂性度量根据运算符计算。每出现一个运算符（或者一对运算符“[]”、“()”），则将该运算符对应的复杂度累加到表达式的复杂度中。

- 选择符：选择符运算有两种，“[]”和“.”。第一种选择符用于数组中元素的选择，第二种选择符用于记录结构中元素的选择。这两种选择符的复杂度均为 2。
- 整型运算：整型运算中加、减运算的复杂度为 2，乘、除、模运算的复杂度为 4。
- 关系运算：关系运算的所有运算符，包括相等、不等、大于、大于等于、小于、小于等于共 6 种运算符，其复杂度均为 4。
- 布尔运算：布尔运算的所有运算符，包括取反、与、或共 3 种运算符，其复杂度均为 6。
- 优先级运算：表达式中的小括号对复杂度为 6。

4.3.7 语句的复杂度

- 赋值语句：每一赋值语句的复杂度基值为 2，再加上表达式的复杂度，就是该赋值语句的复杂度。
- 调用语句：每一调用语句的复杂度基值为 8，再加上实参的复杂度（实参数量不计入复杂度中，但实参的表达式的复杂度需要计入），就是该调用语句的复杂度。
- 分支语句与循环语句：与赋值语句和调用语句不同，分支语句和循环语句是复合语句，且存在嵌套情况。根据嵌套的层数不同，对分支语句和循环语句中子语句的复杂度进行计算，再将结果值进行累加，得出最终的语句复杂度。其计算规则为：对于复合语句中嵌套的子复合语句，仅进行简单累加；而对于复合语句 s 的每一个简单子语句 s' 和表达式 e ，计算公式如下：

$$s = (e + s') * (2^{\text{循环语句层数}}) * (1 + \text{分支语句层数})$$

4.3.8 语句复杂度计算示例

示例 1

设有如下源代码片段：

```
WHILE e1 DO      (* 循环语句层数 = 1, 分支语句层数 = 0 *)
  s1;
  IF e2 THEN      (* 循环语句层数 = 1, 分支语句层数 = 1 *)
    s2;
    WHILE e3 DO    (* 循环语句层数 = 2, 分支语句层数 = 1 *)
      s3;
      IF e4 THEN    (* 循环语句层数 = 2, 分支语句层数 = 2 *)
        s4;
      END;
    END;
  END;
END;
```

其中 $e1$ 、 $e2$ 、 $e3$ 、 $e4$ 分别为表达式， $s1$ 、 $s2$ 、 $s3$ 、 $s4$ 分别为语句（简单语句）。故，该语句的复杂度计算公式为：

$$s = (e4 + s4) * (2^4) * (1 + 4) + (e3 + s3) * (2^3) * (1 + 3) + (e2 + s2) * (2^2) * (1 + 2) + (e1 + s1) * (2^1) * (1 + 1)$$

示例 2

设有如下源代码片段：

```
WHILE x # 0 DO
  x:=x-1;
  IF x = 1 THEN
    y:=10;
    WHILE y#0 DO
      y:=y-1;
      IF y=1 THEN
        WriteLn
      END;
    END;
  END;
END ;
```

该语句存在两个循环语句和两个分支语句的嵌套。最内层的分支语句其循环层数为 2，分支层数也为 2，表达式 $y = 1$ 的复杂度为 4，语句 `WriteLn` 的复杂度为 8，根据公式计算得复杂度为 $(4 + 8) * (2^2) * (1 + 2) = 144$ 。

内层的循环语句（包含如上复杂度为 144 的分支语句）的循环层数为 2，分支层数为 1。含有一个表达式 $y \neq 0$ 的复杂度为 4、一个赋值语句 $y := y - 1$ ；其复杂度为 4、以及一个分支语句（复杂度为 144）。根据计算规则，可得复杂度为 $144 + (4 + 4) * (2^2) * (1 + 1) = 208$ 。

同理，可算出外层分支语句的复杂度为 $208 + (4 + 2) * (2^1) * (1 + 1) = 232$ ，外层的循环语句复杂度为 $232 + (4 + 4) * (2^1) = 248$ 。

4.3.9 其他部分复杂度

- 过程定义：过程定义的复杂度基值为 20，还需要计入过程的形式参数部分、过程的声明部分、以及语句部分的复杂值。
- 形式参数：形式参数部分的复杂度计算包括两个部分：形式参数的数量和形式参数的类型复杂度。每多一个形式参数，其复杂值加一，同时还需计入该形参的类型复杂度。

形式参数的 SDD 规则如下：

formal_parameters : "(" fp_section ")"	$$$ = \$2.val;$
fp_section : fp_section ";" var_if identifier_list ":" type	$$$ = \$1.val + \$4.val + \$6.val;$
fp_section : var_if identifier_list ":" type	$$$ = \$2.val + \$4.val;$

（其非终结符中 identifier_list 在之前的 SDD 规则中有定义）

4.3.10 其他部分复杂度计算示例

设有如下源代码片段：

```
MODULE Sample;
  TYPE intval=INTEGER;
  PROCEDURE Test(VAR x:INTEGER;y,z:INTEGER;VAR m:intval);
  END Test;
BEGIN
  Test(1,2,3);
END Sample.
```

过程 **Test** 中有四个形式参数，三个为整型（但第 2、3 参数使用同一个类型声明，类型复杂值为 1），一个为自定义类型(复杂度为 2)，所以形式参数部分的复杂度为 8。过程定义复杂度基值为 20。语句部分复杂度为 8。所以该源码的复杂度为 $20+8+8=36$ 。

5 实验软装置

为便于同学们开展实验，本实验提供了实验软装置。每位同学应下载该实验软装置，并基于它开展实验过程。

5.1 目录和文件组织

本实验项目提供的实验软装置包括以下文件：

- **exceptions.jar** 文件中存放了所有预定义的异常类。
- **complexity.jar** 中存放了源码复杂性度量的自动化测试工具。
- **java-cup-11b.jar** 文件为 javacup 类库。
- **jflex-full-1.8.2.jar** 文件为 jflex 类库。
- [PENDING]

5.2 异常类设计

本实验要求在词法分析、语法分析、语义分析等阶段发现源程序中的错误时，抛出相应的异常对象。实验软装置中预定义了所有的异常类型，你的分析程序中应根据相应的错误类别抛出最准确的异常对象；如果你认为某类异常还可以细分，则允许利用继承关系派生出更明细的异常类。

5.2.1 异常类程序包

所有异常类均组织在一个名为 **exceptions** 的程序包中，相应的类文件存放在 **exceptions** 子目录中。

5.2.2 通用异常类

为更好地对异常进行分类和组织，本实验项目的实验软装置中设计了几个通用的异常类。通常你的分析程序不应抛出这些通用异常类的对象实例，而应抛出它们的派生类异常对象，以便更准确地反映找到的错误的类别。

OberonException

所有找到的错误对应的异常对象的根（Root）类。

LexicalException

表示找到的错误属于词法错误，是 **OberonException** 的派生类。本文档 5.3.1 小节详细介绍了该类的所有派生异常类。

SyntacticException

表示找到的错误属于语法错误，是 **OberonException** 的派生类。本文档 5.4.1 小节详细介绍了该类的所有派生异常类。

SemanticException

表示找到的错误属于语义错误，是 **OberonException** 的派生类。本文档 0 小节详细介绍了该类的所有派生异常类。

5.3 词法分析实验软装置

本小节介绍的词法分析实验软装置主要供本实验项目中实验二和实验三使用。

5.3.1 词法错误的异常类

本小节所有异常均为 **LexicalException** 的派生类。

IllegalSymbolException

当识别一个单词时遇到不合法的输入符号（譬如@、\$等符号）则抛出该异常。

IllegalIntegerException

当整数常量（无论是十进制还是八进制）与其后的标识符之间无空白分隔时抛出该异常。

IllegalIntegerRangeException

当识别出的整数常量(无论是十进制还是八进制)大于本文档约定的整数常量值最大限制时抛出此异常。

IllegalOctalException

当 0 开头的整数常量中含有 0~7 之外的符号（包括 8 和 9）时抛出该异常。

IllegalIdentifierLengthException

当识别出的一个标识符长度超过最大限制时抛出该异常。

MismatchedCommentException

当 “(” 开头的注释直至扫描到最后一个符号都找不到配对的 “)” 时抛出该异常。

5.3.2 词法分析主程序

本实验软装置提供了语法分析的软设置，你可以参考下面的程序并在实验 2 中以此为基础完善此法分析主程序。对于 Oberon 源码数据输入部分和结果的读取方法你可以自由发挥。

主程序简单地根据读入的测试用例字符串，调用你的词法分析入口程序完成单词扫描过程，再根据返回结果判断是否与预期输出一致。其源代码如下：

```
public static void main(String[] args) throws LexicalException, IOException {
    // 源码数据输入与正确结果（可以使用其他方式读取）
    String code="MODULE Sample;\n\tVAR a:INTEGER;\n\tPROCEDURE Test;\n\tEND
Test;\nBEGIN\n\tRead(a);\nEND Sample.";
    StringReader reader=new StringReader(code);
    String answer="#1) (#8)[Sample] (#5) (#6) (#8)[a] (#7) (#9) (#5) (#2)
(#8)[Test] (#5) (#4) (#8)[Test] (#5) (#3) (#8)[Read] (#11) (#8)[a] (#12) (#5)
(#4) (#8)[Sample] (#10) (#0)";

    // 创建生成的 Scanner 类实例
    SampleScanner scanner=new SampleScanner(reader);
    StringBuilder builder=new StringBuilder();
    while(!scanner.yyatEOF()){// 获取 Scanner 输出的 token
        java_cup.runtime.Symbol s=scanner.next_token();
        // 记录 token
        if(s.value!=null){
            builder.append("(" + s.toString() + ")[" + (String)s.value + "]);");
        }else{
            builder.append("(" + s.toString() + ")");
        }
        builder.append(" ");
    }
    // 比较 Scanner 的输出和正确结果
    String result=builder.toString().trim();
    if(answer.equals(result)){
        System.out.println("check success");
    }else{
        System.out.println("check fail");
    }
}
```

5.3.3 词法分析程序测试用例

本实验设置没有提供自动化词法分析程序的测试用例，需要你完成这部分测试工作，包括测试的流程和测试样例。保证测试用例能够尽可能覆盖大部分情况和异常。测试流程你可以参考上述的 scanner 使用例子。

5.4 语法分析和语法制导翻译实验软装置

本小节介绍的语法分析和语法制导翻译实验软装置主要供本实验项目中实验三和实验四使用。

5.4.1 语法错误的异常类

本小节所有异常均为 **SyntacticException** 的派生类。

MissingLeftParenthesisException

当分析到左右圆括号不匹配、且缺少左括号时则抛出该异常。

MissingRightParenthesisException

当分析到左右圆括号不匹配、且缺少右括号时则抛出该异常。

MissingOperatorException 异常

当分析到程序中缺少运算符时，或调用预定义函数缺少相应的参数时则抛出该异常。

MissingOperandException 异常

当分析到程序中缺少操作数时则抛出该异常。

5.4.2 语义错误的异常类

本小节所有异常均为 **SemanticException** 的派生类。

TypeMismatchedException

当分析到表达式、赋值语句、或参数传递等构造中出现类型不兼容错误时则抛出该异常。

ParameterMismatchedException

当分析到过程调用的实际参数数目与过程声明的形式参数数目不一致时则抛出该异常。注意，当实际参数的数目与形式参数一致，只是存在某一参数的类型不兼容时，应抛出 **TypeMismatchedException** 异常而不是抛出本异常。

5.5 自动化测试 API

本实验内置了 **ALIOTH** 工具的自动化测试工具，该测试工具已经打成 jar 包放在了实验软装置的 lib 目录下，你通过 “import complexity.*;” 使用该测试工具。

测试工具包含四个文件 **ComplexityTest** 类、**Calculator** 接口以及 **simple_testcase.xml** 和 **full_testcase.xml** 两个测试文件。

使用该自动化测试工具的流程如下：

- 首先你需要实现 **Calculator** 接口，该接口只有一个函数声明 **calculate()**，其功能为计算一个 Oberon 源程序的复杂度，该函数的参数为给定的 Oberon-0 源代码字符串，返回值是计算出的源程序复杂度。
- 然后创建 **ComplexityTest** 类的一个实例（创建该实例需传入一个已实现的 **Calculator** 实例），该类实例可以调用 **simpleTest()** 方法、**fullTest()** 方法以及 **test()** 方法进行测试。

- **simpleTest()**方法，方法读取 **simple_testcase.xml** 文件进行简单测试。测试文件中有一个测试样例，且结果值为 0。
- **fullTest()**方法，方法读取 **full_testcase.xml** 文件进行测试，测试文件包含完整的测试用例（包括对数据声明、过程声明、表达式、语句等功能的复杂度计算），在方法结束后使用 **getRate()**方法输出测试的正确率。
- **test()**方法，需要传入一个 **.xml** 文件的输入流（**InputStream**），通过该方法，你可以将自定义的测试文件传入进行自动化测试（实验软装置中没有提供异常测试，你需要使用自定义测试文件增加测试用例，测试用例越多你的测试用例部分得分就越高）。

测试工具的使用示例：

```
import complexity.Calculator;
import complexity.ComplexityTest;

public class ComplexityDemo implements Calculator{
    public static void main(String[] args) throws Exception {
        ComplexityTest tester= new ComplexityTest(new ComplexityDemo());
        //简单测试
        tester.simpleTest();
        //使用完整用例测试
        tester.fullTest();
        //获取测试成功率
        double rate=tester.getRate();
    }

    @Override
    public String calculate(String arg0) {
        // TODO 实现复杂度的计算 arg0 为Oberon 程序源码字符串
        return "0";
    }
}
```

测试文件的格式如下：

```
<?xml version="1.0"?>
<definitions>
  <!-- 简单测试 -->
  <item>
    <id>0001</id>
    <desc>SimpleTest</desc>
    <input>
      MODULE Sample;
      END Sample.
    </input>
    <output>0</output>
  </item>
</definitions>
```

