
编译原理课程项目 3

计算机学院 王欣明

2019 春季学期

本实验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. 编程语言

Java 语言，JDK 1.5 以上 版本。

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，提交可执行的批处理文件。

3. 编码规范

学生在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 Java 程序设计语言的编码规范，参见：<http://www.oracle.com/technetwork/java/codeconvtoc-136057.html>。完成后的代码应使用 JDK 附带的文档工具 `javadoc` 根据源程序中的文档化注释生成相应的文档。

4. 实验内容：Oberon-0 程序流程图生成器

本实验是编译原理实验环节的一个综合型、应用型实验。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 的逆向工程工具，根据一个输入的 Oberon-0 源程序自动绘制每个过程的程序流程图。

本实验借助于工具的设计与实现，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。

本实验分解为3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。

4.1. 子实验一：熟悉Oberon-0 语言定义

附录部分给出了 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，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。

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。

步骤 1.1 讨论 Oberon-0 语言的特点

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。

步骤 1.2 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在 Oberon-0 语言中并未出现？

提交结果

将子实验一的所有结果存放在子目录 **xxxxxxxxxNNN\ex1** 中，其中 **xxxxxxxxx** 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。

步骤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、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**，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。

评价标准

项 目	权重 (%)
1、一个正确的Oberon-0 源程序	30
2、Oberon-0 语言特点的讨论	30
3、Oberon-0文法二义性的讨论	30
4、文档组织及readme 文件等	10

4.2. 子实验二：生成词法分析程序（JFlex）

子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。

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。

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，并写成正则定义式的形式。

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/jflex-1.4.1.zip> 可下载该工具的最新稳定版本1.4.1;如果你因网络原因无

法访问该网 站，可转向 <http://sourceforge.net/projects/jflex/> 下载相应版本 JFlex，或直接从本课程教学网站下载该版本 JFlex。这一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。

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。

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 程序，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。

提交结果

将步骤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，命名规则同步骤一。

步骤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。

自动生成的 Oberon-0 语言词法分析程序，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`。

- 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。

评价标准

项 目	权重 (%)
1、Oberon-0 语言词法规则的正则定义式	30
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、文档组织及readme文件等	10

4.3. 子实验三：生成语法分析程序（JavaCUP）

子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。

步骤3.1、下载自动生成工具JavaCUP

实验三选用由美国卡内基·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。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。

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。

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，自动绘制出不同模块中每个函数（也叫过程）的流程图。本文档第 6 节部分详细介绍了流程图的定义和画图的 API。

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

提交结果

将子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，命名规则同子实验一。

子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、完成日期、以及其他补充说明。
- 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\javacup** 子目录中存放你所使用的 JavaCUP 工具。

评价标准

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	40
3、文档组织及readme 文件等	10

5. 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。

5.1. 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。

5.2. 词法定义

Oberon-0 语言定义了非常简单的语法规则。

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。

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**。

5.3. 语法定义

本小节以EBNF 定义了Oberon-0 语言的形式语法。

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)。

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 ;
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} ;

5.4. 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。

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。

运算符与表达式

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。

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;
```

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：“.”“[]”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;
  ...
```

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。

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。

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。

5.5. 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。该模块中包含了几个简单的过程，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.

```

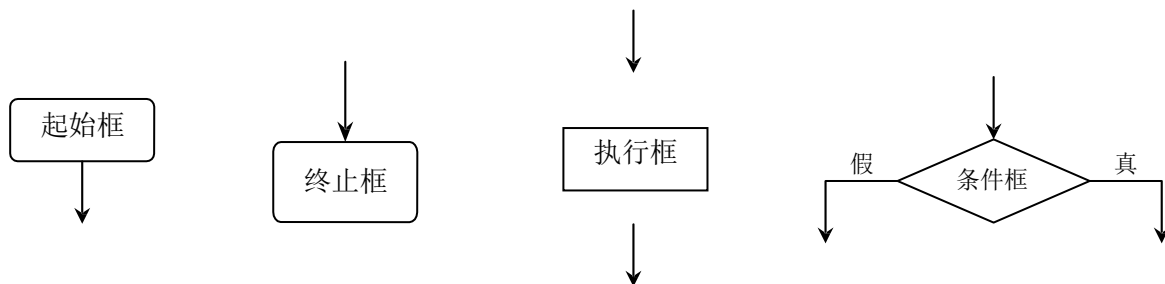
6. 流程图

本实验项目要求开发的软件工具可根据一个Oberon-0 语言源程序自动绘制出一个模块中所有过程的流程图。

6.1. 流程图定义

流程图（Flowchart）又称程序流程图、程序框图，是一个子程序控制流设计的图形表示，其中展示了 程序执行过程中可能遍历的所有路径。

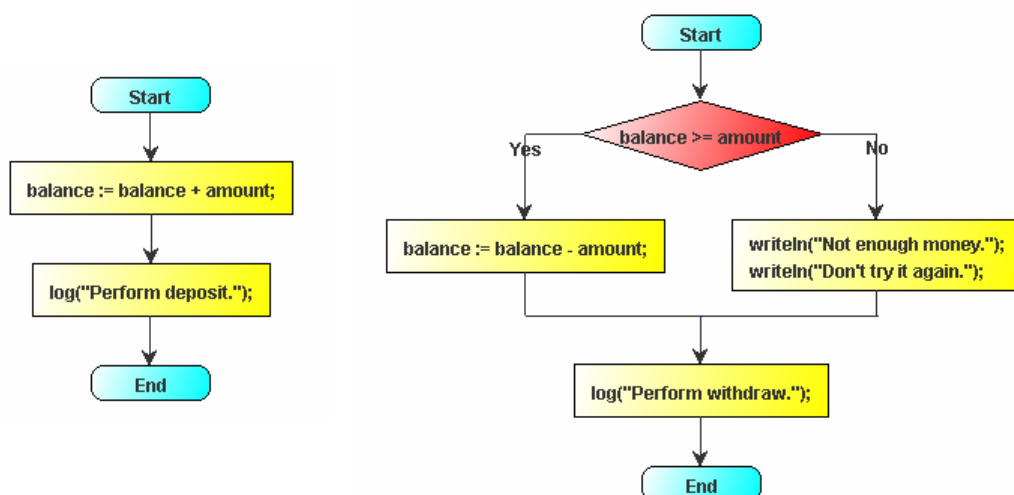
流程图中具有以下基本图形：



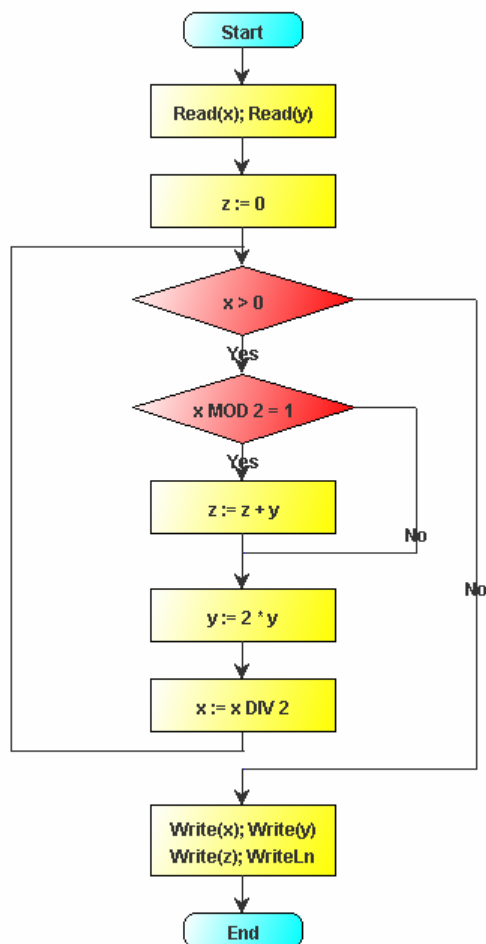
关于流程图的更详细资料可参阅：<http://en.wikipedia.org/wiki/Flowchart>。

6.2. 流程图例子

对于一个 Account 模块中的存款和取款过程，生成的两个流程图例子如下图所示。



对于本文档 5.5 节例子程序的Multiply 过程，流程图例子如下图所示。



6.3. 画流程图的框架代码

实验框架代码提供了预定义的绘图代码，同学们可以调用这些现成的代码绘制图形。框架代码的实现是基于开源项目 Jgraph。

要绘制第三章3.5 小节的Oberon 源程序定义Sample 模块中3 个过程对应的流程图,可按如下方式调用 预定义的API（意即你的翻译模式应按如下次序执行其中的语义运作）:

```

import flowchart.*;
public class FlowchartDemoOberon {
    public static void main(String[] args) throws Exception {
        Module sampleModule = new Module("Sample");
        {
            Procedure proc = sampleModule.add("Multiply");
            proc.add(new PrimitiveStatement("Read(x); Read(y)"));
            proc.add(new PrimitiveStatement("z := 0"));
            WhileStatement wstmt = new WhileStatement("x > 0");
            proc.add(wstmt);
            {
                IfStatement istmt = new IfStatement("x MOD 2 = 1");

```

```

        wstmt.getLoopBody().add(istmt);
    {
        istmt.getTrueBody().add(new PrimitiveStatement("z := z + y"));
    }
    wstmt.getLoopBody().add(new PrimitiveStatement("y := 2 * y"));
    wstmt.getLoopBody().add(new PrimitiveStatement("x := x DIV 2"));
}
proc.add(new PrimitiveStatement(
    "Write(x); Write(y)<br>Write(z); WriteLn"));
}

Procedure proc = sampleModule.add("Divide");
proc.add(new PrimitiveStatement("Read(x); Read(y)"));
proc.add(new PrimitiveStatement("r := x; q := 0; w := y"));
WhileStatement wstmt = new WhileStatement("w <= r");
proc.add(wstmt);
{
    wstmt.getLoopBody().add(new PrimitiveStatement("w := 2 * w"));
}
wstmt = new WhileStatement("w > y");
proc.add(wstmt);
{
    wstmt.getLoopBody().add(new PrimitiveStatement("q := 2 * q"));
    wstmt.getLoopBody().add(new PrimitiveStatement("w := w DIV 2"));
    IfStatement istmt = new IfStatement("w <= r");
    wstmt.getLoopBody().add(istmt);
    {
        istmt.getTrueBody().add(new PrimitiveStatement("r := r - w"));
        istmt.getTrueBody().add(new PrimitiveStatement("q := q + 1"));
    }
}
proc.add(new PrimitiveStatement(
    "Write(x); Write(y)<br>Write(q); Write(r); WriteLn"));
}

Procedure proc = sampleModule.add("BinSearch");
proc.add(new PrimitiveStatement("Read(n)"));
proc.add(new PrimitiveStatement("k := 0"));
WhileStatement wstmt = new WhileStatement("k < n");
proc.add(wstmt);
{
    wstmt.getLoopBody().add(new PrimitiveStatement("Read(a[k])"));
    wstmt.getLoopBody().add(new PrimitiveStatement("k := k + 1"));
}
proc.add(new PrimitiveStatement("Read(x)"));
proc.add(new PrimitiveStatement("i := 0; j := n"));
wstmt = new WhileStatement("i < j");
proc.add(wstmt);

```

```
{
  wstmt.getLoopBody().add(new PrimitiveStatement("k := (i + j) DIV 2"));
  IfStatement istmt = new IfStatement("x < a[k]");
  wstmt.getLoopBody().add(istmt);
  {
    istmt.getTrueBody().add(new PrimitiveStatement("j := k"));
    istmt.getFalseBody().add(new PrimitiveStatement("i := k + 1"));
  }
}
proc.add(new PrimitiveStatement(
  "Write(i); Write(j)<br>Write(a[j]); WriteLn"));
}
sampleModule.show();
}
```
