

****

# 编译原理实践

——PL/0编译系统

**姓名：柏林岑**

**学号：16211101**

**学院：软件学院**

目录

[编译原理实践 1](#_Toc533348272)

[——PL/0编译系统 1](#_Toc533348273)

[一、实验目的 3](#_Toc533348274)

[1.1具体任务 3](#_Toc533348275)

[1.2实验目标 3](#_Toc533348276)

[二、PL/0编译系统的设计分析 3](#_Toc533348277)

[2.1 PL/0文法概述 3](#_Toc533348278)

[2.2 PL/0文法说明 4](#_Toc533348279)

[2.2.2 PL/0语法图 5](#_Toc533348280)

[2.3 PL/0编译器的系统结构 6](#_Toc533348281)

[三、PL/0编译系统的实现 7](#_Toc533348282)

[3.0程序结构 7](#_Toc533348283)

[3.1词法分析器(LexicalAnalysis) 8](#_Toc533348284)

[3.1.1词法分析器的功能 8](#_Toc533348285)

[3.1.2词法分析器的设计 8](#_Toc533348286)

[3.1.3核心函数说明 9](#_Toc533348287)

[3.2符号表管理（SymbolTable） 9](#_Toc533348288)

[3.2.1符号表的功能 9](#_Toc533348289)

[3.2.2核心函数说明 10](#_Toc533348290)

[3.3语法分析器（SyntaxAnalysis） 10](#_Toc533348291)

[3.3.1语法分析器的介绍及功能 10](#_Toc533348292)

[3.3.2核心函数说明 11](#_Toc533348293)

[3.4 目标代码（Pcode） 11](#_Toc533348294)

[3.4.1目标代码的功能 11](#_Toc533348295)

[3.5解释器（Interpreter） 13](#_Toc533348296)

[3.5.2核心函数说明 13](#_Toc533348297)

[3.6错误处理（Errors） 14](#_Toc533348298)

[3.6.1错误处理的功能 14](#_Toc533348299)

[3.6.2错误处理的原则 14](#_Toc533348300)

[3.6.3错误信息表 14](#_Toc533348301)

[四、PL/0编译系统的GUI界面 15](#_Toc533348302)

[五、测试报告 18](#_Toc533348303)

[六、实验感想 21](#_Toc533348304)

[七、参考资料 22](#_Toc533348305)

## 一、实验目的

### 1.1具体任务

文件输入：符合PL/0文法的源程序（自己要有5个测试用例，包含出错的情况，还要用老师提供的测试用例进行测试）

输出：P-Code

错误信息：参见教材第316页表14.4（新书第411页表17.4）。

错误信息尽量详细（行号，错误类型）

P-Code指令集：参见教材第351页表15.14。

语法分析部分要求统一使用递归下降子程序法实现。

编程语言自定，可使用C、C++、C#或Java等。（尽量选用有可执行文件的）过程。

加分项（选做）：

解释程序

良好的GUI界面

### 1.2实验目标

《编译原理》课程设计是编译原理课程必不可少的一个环节，通过课程设计，加深对编译原理的教学内容的了解，以及实现编译原理各部分知识的融合。进而提高学生分析问题、解决问题，从而运用所学知识解决实际问题的能力。

在分析理解一个教学型编译程序（如PL/0）的基础上，对其词法分析程序、语法分析程序和语义处理程序进行实现，达到进一步了解程序编译过程的基本原理和基本实现方法的目的。

## 二、PL/0编译系统的设计分析

### 2.1 PL/0文法概述

PL/0语言是一种类PASCAL语言，是教学用程序设计语言，它比PASCAL语言简单，作了一些限制。PL/0的程序结构比较完全，赋值语句作为基本结构，构造概念有：

- 顺序执行、条件执行和重复执行，分别由begin/end,if then else和while do语句表示。

- PL0还具有子程序概念，包括过程说明和过程调用语句。

- 在数据类型方面，PL0只包含唯一的整型，可以说明这种类型的常量和变量。

- 运算符有+，-，\*，/，=，<>，<，>，<=，>=，(，)。

- 说明部分包括常量说明、变量说明和过程说明

PL/0过程没有参数，但可以递归调用，因此，过程所加工的数据只能通过全局变量进行传递。

### 2.2 PL/0文法说明

<程序> ::= <分程序>.

<分程序> ::= [<常量说明部分>][变量说明部分>][<过程说明部分>]<语句>

<常量说明部分> ::= const<常量定义>{,<常量定义>};

<常量定义> ::= <标识符>=<无符号整数>

<无符号整数> ::= <数字>{<数字>}

<标识符> ::= <字母>{<字母>|<数字>}

<变量说明部分>::= var<标识符>{,<标识符>};

<过程说明部分> ::= <过程首部><分程序>{<过程说明部分>};

<过程首部> ::= procedure<标识符>;

<语句> ::= <赋值语句>|<条件语句>|<当型循环语句>|<过程调用语句>|<读语句>|<写语句>|<复合语句>|<重复语句>|<空>

<赋值语句> ::= <标识符>:=<表达式>

<表达式> ::= [+|-]<项>{<加法运算符><项>}

<项> ::= <因子>{<乘法运算符><因子>}

<因子> ::= <标识符>|<无符号整数>|'('<表达式>')‘

<加法运算符> ::= +|-

<乘法运算符> ::= \*|/

<条件> ::= <表达式><关系运算符><表达式>|odd<表达式>

<关系运算符> ::= =|<>|<|<=|>|>=

<条件语句> ::= if<条件>then<语句>[else<语句>]

<当型循环语句> ::= while<条件>do<语句>

<过程调用语句> ::= call<标识符>

<复合语句> ::= begin<语句>{;<语句>}end

<重复语句> ::= repeat<语句>{;<语句>}until<条件>

<读语句> ::= read'('<标识符>{,<标识符>}')‘

<写语句> ::= write'('<标识符>{,<标识符>}')‘

<字母> ::= a|b|...|X|Y|Z

<数字> ::= 0|1|2|...|8|9

Tips：

数据类型：无符号整数

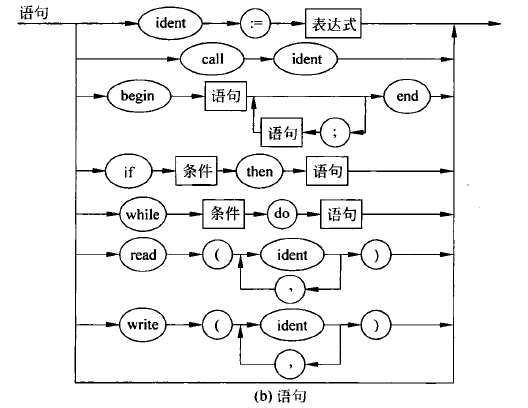
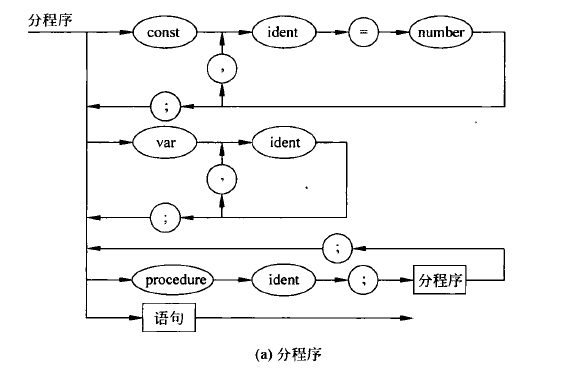
标识符类型：简单变量(var)和常数(const)

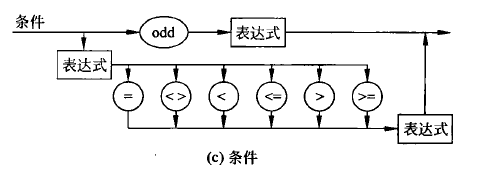
数字位数：小于14位

标识符的有效长度：小于10位

过程嵌套：小于3层

#### 2.2.2 PL/0语法图





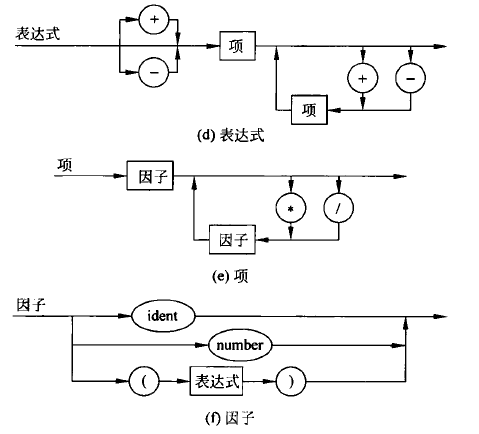


图1 PL/0语法图

### 2.3 PL/0编译器的系统结构

PL/0编译系统是一个编译-解释执行程序,整个编译过程分两个阶段进行。第一阶段先把PL/0源程序编译成假想计算机的目标(P-code指令)程序,第二阶段再对该目标程序进行解释执行,得到运行结果。PL/0编译程序采用一遍扫描，即以语法分析为核心,由它调用词法分析程序取单词,在语法分析过程中同时进行语义分析处理,并生成目标指令。如遇语法、语义错误,则随时调用出错处理程序,打印出错信息。在编译过程中要利用符号表的登录和查找来进行信息之间的联系。一遍扫描的PL/0编译和P-code解释执行框图如图1所示：

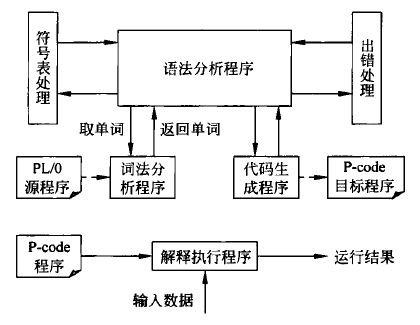


图2 PL/0编译系统结构框图

## 三、PL/0编译系统的实现

### 3.0程序结构

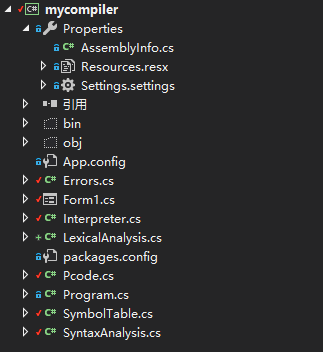


图3 程序的文件组织图

整个PL/0编译系统主要由词法分析器（LexicalAnalysis），符号表（SymbolTable），语法分析器（SyntaxAnalysis），目标代码（Pcode），解释器（Interpreter），错误处理（Errors）以及GUI设计（Form1）7个部分组成，接下来将对其具体实现以及函数功能进行说明：

### 3.1词法分析器(LexicalAnalysis)

#### 3.1.1词法分析器的功能

将PL/0编译系统中所有的字符，字符串的类型按如下表格分类：

**表1 PL/0编译系统字符/字符串类型**

|  |  |
| --- | --- |
| 保留字 | Begin,end,if,then,else,const,procedure,var,  do,while,odd,call,read,write,repeat,until |
| 算数运算符 | + | - | \* | / |
| 比较运算符 | <> | < | <= | > | >= | = |
| 赋值符 | := | = |
| 标识符 | 变量名，过程名，常数名 |
| 常数 | 整数 |
| 界符 | , | . | ; | ( | ) |

词法分析器通过扫描源程序，按语言的词法规则识别出各类单词符号（Token），并将有关字符组合成为单词并保存入单词表，同时进行词法检查。

#### 3.1.2词法分析器的设计

词法分析器的主要工作流程如下：

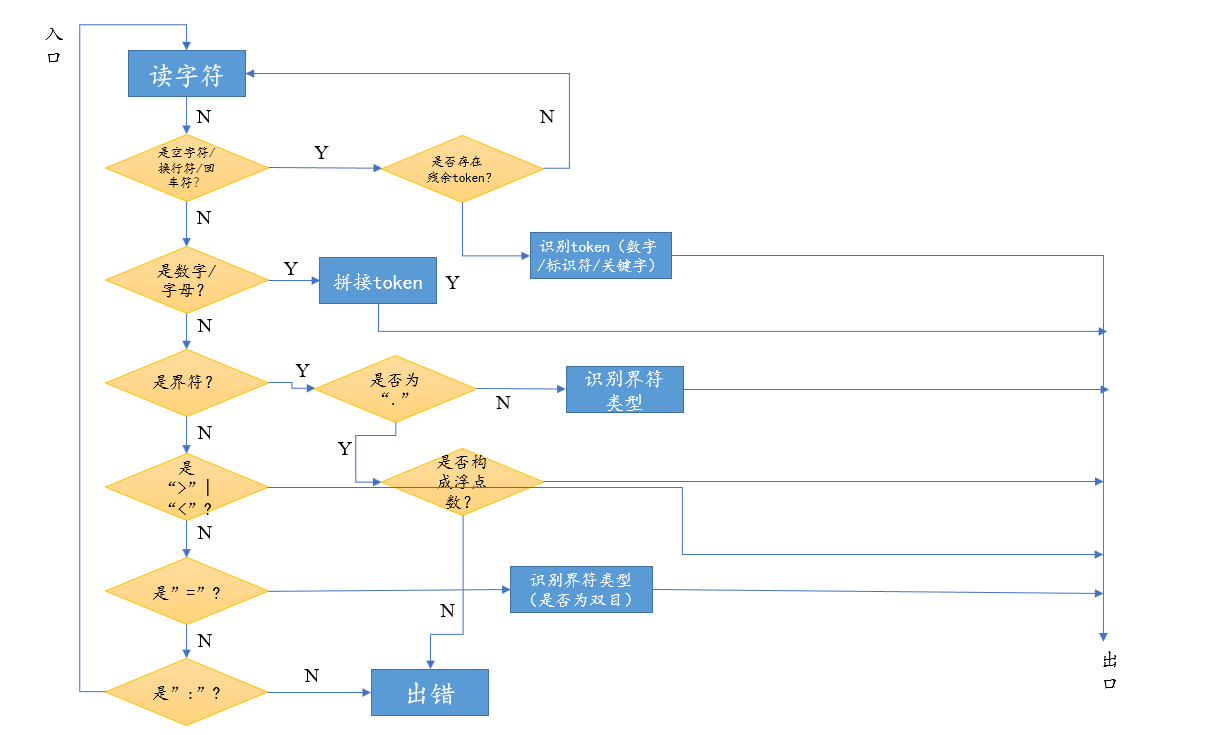


图4 Lexanaly工作流程示意图

（1）在主函数LexAnaly中每次调取一行源程序，通过循环依次取得单个字符ch；

（2）跳过源程序中的空格、缩进、换行等字符，对之前的token进行Judge判断；

（3）对字母和数字类型的ch拼接形成token，直到读到其它类型的ch，调用Judge判断；

（4）Judge通过查询关键字表以及整数类型转换函数，对当前token进行判断（无符号整数/浮点数/关键字/标识符） 然后存入单词表Word；

（5）对于界符以及运算符，根据其类型分别以其对应枚举类型的方式存入单词表Word；

（6）PL/0的语法分析采用了“单符号先行”技术，在进入某个语法成分的分析子程序之前，先读取一个单词放入tmpword中，但由于主程序中通过循环对源程序的遍历识别，在进行语法分析前，已经构造好了单词表Word，在语法分析器中被调用的Getsym其实是对单词表的依次查询。

#### 3.1.3核心函数说明

**表2 词法分析器核心函数说明**

|  |  |
| --- | --- |
| 函数名 | 函数功能 |
| Insert | 添加单词表成员（单词识别码、单词内容、行信息） |
| Judge | 在Lexanaly中被调用实现对token进行判断（无符号整数/关键字/标识符） 然后存入单词表 |
| reserver | 识别关键字，并且返回识别码 |
| Lexanaly | 按行读取单词，调用各类判断函数判断其类型，并调用Insert添加单词 |

### 3.2符号表管理（SymbolTable）

#### 3.2.1符号表的功能

符号表项Item的定义如下：

public struct Item

{

public String name; //记录的名字

public int type; //记录的类型,有3种类型：constant(0), variable(1), procedure(2)

public int value; //记录的值

public int level; //记录所在的层

public int address; //记录的地址

public int size; //需要分配的空间，当记录类型为procedure时使用，默认值为0 };

符号表每项记录中的level域应填人说明该变量名或过程名的分程序的层次。主程序的层次为0,各嵌套分程序的层次随嵌套深度递增(PL/0程序限制level的最大值为3),符号表中的adr域应填人每层局部变量所分配单元的相应地址(其起始值为3,由地址分配索引变量dx指定,后面将详述)。对过程名,则应填人编译该过程所生成的P-code指令序列的人口地址。PL/O规定所有名字必须先说明或定义后才能引用。

符号表相关操作主要有两种：

1. 符号表项的增加：TableFill，将该标识符及相关的信息插入标识符表中。
2. 符号表的查询，查找标识符在符号表中的位置从tx开始倒序查找标识符，找到则返回在符号表中的位置，否则返回-1。

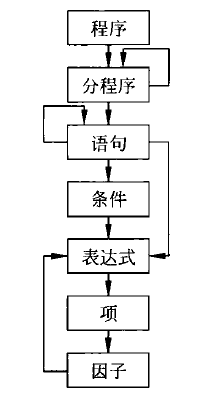
#### 3.2.2核心函数说明

**表3 符号表核心函数说明**

|  |  |
| --- | --- |
| 函数名 | 函数功能 |
| TableFill | 向符号表中添加一条记录 |
| Position | 查找标识符在符号表中的位置 |
| SetAdd | 设置记录表项的地址（Addr） |

### 3.3语法分析器（SyntaxAnalysis）

#### 3.3.1语法分析器的介绍及功能

PL/0采用了**递归子程序法**进行语法分析，即为每一个语法成分编写一个分析子程序，根据当前读取的符号，可以选择相应的子程序进行语法分析。采用不带回溯的递归子程序法，对语言文法有一定的要求：

1. 文法必须是非左递归；
2. 文法的任一非终结符，其规则右部多个选择所生成的FIRST集合两两不相交；
3. 若文法具有形如，则。

根据PL/0的文法BNF表达，下表中列出了PL/0有关的语法成分的FIRST集合FOLLOW集，可以判断出满足递归子程序的要求： 图5 PL/0依赖图

**表4 PL/0的头符号和后继符号集**

|  |  |  |
| --- | --- | --- |
| 非终结符 | FIRST集 | FOLLOW集 |
| 分程序block | const var procedure id if call begin while repeat read write | . ; |
| 语句statement | Id call begin if while read write until | . ; end |
| 条件condition | odd + - ( id num | then do |
| 表达式expression | + - ( id num | , ; ) R end then do |
| 项term | id num ( | , ; ) R + - end then do |
| 因子factor | id num ( | . ; ) R + - \* / end then do |

#### 3.3.2核心函数说明

结合PL/0文法的语法图（图1）以及依赖图（图5）可以编写出使用递归下降子程序法实现的语法分析器，以下是其核心函数的介绍：

**表5语法分析器核心函数说明**

|  |  |
| --- | --- |
| 函数名 | 函数功能 |
| Block | 分程序模块 |
| ConstDeclaration | 常量声明模块 |
| VarDeclaration | 变量声明模块 |
| Condition | 条件模块 |
| Expression | 表达式模块 |
| Term | 项模块 |
| Factor | 因子模块 |
| Statement | 语句模块（包含8个子语句模块） |
| Test | 检测当前符号合法性，并进行跳读 |

### 3.4 目标代码（Pcode）

#### 3.4.1目标代码的功能

Pcode语言：一种栈式机的语言。此类栈式机没有累加器和通用寄存器，有一个栈式存储器，有四个控制寄存器（指令寄存器 I，指令地址寄存器 P，栈顶寄存器 T和基址寄存器 B），算术逻辑运算都在栈顶进行。

指令格式形如F L A，其中F表示操作码，可用枚举变量表示；L表示层次差 （标识符引用层减去定义层）；A不同的指令含义不同。对于不同的Pcode指令，各指令含义如下：

**表6 PCode指令含义对照表**

|  |  |
| --- | --- |
| 指令 | 含义 |
| LIT 0, a | 取常量a放到数据栈栈顶 |
| OPR 0,0 | 函数调用结束后的返回 |
| OPR 0,1 | 栈顶元素取反 |
| OPR 0,2 | 次栈顶与栈顶相加，退两个栈元素，相加值进栈 |
| OPR 0,3 | 次栈顶减栈顶 |
| OPR 0,4 | 次栈顶乘以栈顶 |
| OPR 0,5 | 次栈顶处以栈顶 |
| OPR 0,6 | 栈顶元素的奇偶判断 |
| OPR 0,7 | 次栈顶对栈顶取模，退两个栈元素，余数进栈 |
| OPR 0,8 | ==判断相等 |
| OPR 0,9 | !=判断不相等 |
| OPR 0,10 | <判断小于 |
| OPR 0,11 | >=判断大于等于 |
| OPR 0,12 | >判断大于 |
| OPR 0,13 | <=判断小于等于 |
| LOD 0,a | 将变量值放到栈顶，a是存储空间，l是调用层与说明层的层次差 |
| STO 0,a | 将栈顶内容送入某变量单元中 ，a是存储空间 |
| CAL 0,a | 调用过程,a是被调用过程的目标程序入口地址 |
| INT 0,a | 在运行栈中为被调用的过程开辟a个单元的数据区 |
| JMP 0,a | 无条件跳转至a地址 |
| JPC 0,a | 条件跳转，当栈顶布尔值非真则跳转至a地址，否则顺序执行（假转） |
| WRT 0,a | 栈顶值输出至屏幕 |
| RED 0,a | 读入一个输入置于栈顶 |

在Pcode.cs中只对pcode指令表项Item以及操作码的枚举类型进行了定义，具体的Pcode生成由语法分析器实现，具体的解释操作由解释器实现。

### 3.5解释器（Interpreter）

3.5.1解释器的功能

假想的一个PL/0计算机，可看作由两个存储器、一个指令寄存器和三个地址寄存器组成。数组pcodeList存放程序Pcode指令集，数组sta作为运行栈， 实现栈式动态存储分配。指令寄存器currentCode存放当前执行的Pcode指令；栈顶地址寄存器sp作为栈顶指针，总是指向运行栈sta的栈顶；程序地址寄存器pc存放下一条要执行的指令地址（数组索引）；基地址寄存器bp，存放当前运行的分程序数据区在数据栈S中的起始地址，即bp总是指向动态链的链头。

PL/0每个过程有自己的局部变量,而采用栈式动态存储分配不可能在编译时就知道该变量在数据栈中的绝对地址。因此,编译时只能确定该变量在它所在分程序数据区中的相对位置,即它相对于本分程序数据区的起始地址(基地址)的位移。考虑到在嵌套分程序的内分程序中可以引用外分程序或主程序中定义的变量,所以还必须在各分程序数据区中设立一个SL(静态链)单元,用来保存它的直接外层分程序数据区的基地址,以便在引用外层说明的变量时,可通过静态链来找到该变量在S数据栈中的确切位置。

在编译时,凡生成的涉及存、取变量的指令(如LOD、STO,RED)时,其l值为引用该变量的分程序层次与说明该变量的分程序层次之差。这样,只要按层次差值l,沿着静态链往回找1次,即可找到说明该变量的分程序数据区的基地址,再加上相对地址值a,就确定了该交量在数据栈S中的“绝对”位置。

在经过语法语义分析后,Pcode的生成是非常直观的,有关表达式的处理都按照逆波兰表示法(后缀表达式)生成相应的运算指令。gen 过程只是简单地把牛成的P code指令送人code指令存储区。cx 是指令索引指针,用来表示下一条要生成的指令的地址。在生产某些转移指令(如JMP.JPC)时,往往不能马上确定转移地址a的值,这时可以把这条指令的cx值保持起来,等后面确定了转移地址以后,再根据所保持的cx值把转移地址返填回去。

由于PL/0程序具有嵌套的分程序结构,所以编译进人分程序后生成的第一条指令为无条件转移指令(JMP)。这条指令是为了跳过它所包含的其他分程序所生成的指令序列，而直接进入程序体中语句部分所生成的指令。语句部分生成的指令序列的第一条指令为INT0,dx,其作用是实现在数据栈S中为本分程序的数据区动态分配存储空问。

#### 3.5.2核心函数说明

**表7 符号表核心函数说明**

|  |  |
| --- | --- |
| 函数名 | 函数功能 |
| Interpret | 根据PCode每条指令的含义，对运行栈进行操作 |
| AddPcode | 在语法分析器中被调用，添加一条新的Pcode指令加入PcodeList |
| Base | 通过过程基址求上一层过程基址 |

### 3.6错误处理（Errors）

#### 3.6.1错误处理的功能

对于简单的错误，如遗忘了分号，赋值符号写错等，除了给出错误信息，可以予以简单校正，使程序尽可能继续进行。在每个语法分析子程序的入口或出口处，调用isLegal函数, 检测下一个取得的单词是否是当前语法成分的合法头符号集或者后继符号，若不是，给出错误信息，并且跳读一段程序，直至取得合法的后继符号。然而，为了防止跳读的程序段太多，程序中设置了一个停止符号集合，此集合中元素为明显可以使程序开始正常编译的符号，所以只要新取来的符号属于合法后继符号集或者停止符号集，都可以停止跳读。

#### 3.6.2错误处理的原则

1. 任何源程序输入序列都不会导致编译工作的崩溃；
2. 尽可能多地发现源程序中的语法和语义在错误，并尽可能准确指出错误位置和错误属性；
3. 尽可能进行校正，对于出现的错误，予以校正使编译继续进行。

#### 3.6.3错误信息表

在教材26个PL/0编译程序的错误编码及出错信息的基础上扩充了部分错误信息，如下表所示：

**表8 PL/0编译程序的错误信息**

|  |  |
| --- | --- |
| 出错编号 | 出错原因 |
| 1 | 常数说明中应是'='而不是':=' |
| 2 | 常数说明中'='后应为整数 |
| 3 | 常数说明中标识符后应为'=' |
| 4 | const,var,procedure后应为标识符 |
| 5 | 漏掉’,’或’;’ |
| 6 | 过程说明后的符号不正确（应是语句开始符或过程定义符） |
| 7 | 应为语句开始符号 |
| 8 | 程序体内语句部分后的符号不正确 |
| 9 | 程序结尾丢了句号 |
| 10 | 语句之间漏了分号 |
| 11 | 标识符未说明 |
| 12 | 赋值语句中，赋值号左部标识符属性应是变量 |
| 13 | 赋值语句中应为赋值运算符':=' |
| 14 | call后应为标识符 |
| 15 | call后标识符属性应为过程,不可调用常量或变量 |
| 16 | 条件语句中缺失then |
| 17 | 丢了end或’;’ |
| 18 | while型循环语句中缺失do |
| 19 | 语句后的符号不正确 |
| 20 | 应为关系运算符 |
| 21 | 表达式内不可有过程标识符 |
| 22 | 表达式中漏掉右括号 |
| 23 | 因子后的非法符号 |
| 24 | 表达式不能以此符号开始 |
| 25 | 不存在的操作符 |
| 26 | 结束符只能出现在程序结尾 |
| 27 | 不能在变量声明语句中尝试给变量赋值 |
| 30 | 这个数太大 |
| 31 | read括号内应该是变量标识符 |
| 33 | 缺少until |
| 35 | ‘:’需要和’=’一起使用给变量赋值 |
| 40 | 应为左括号 |
| 41 | 标识符不能以数字开头 |
| 42 | 结束符出现在了不适当的地方 |
| 43 | 输入了无法被识别的符号 |

在语法分析器中如果识别到错误，则调用AddError函数更新错误信息errorMessage以及错误计数errorcount,最后在页面中检查错误计数，如果大于0则编译过程存在错误，显示错误信息，否则则可以进行下一步的运行操作。

## 四、PL/0编译系统的GUI界面

界面中左上文本框为【代码编辑区】，可直接在【代码编辑区】中输入或编辑PL/0代码；右边是【目标代码（Pcode）区】。

- 最右边有4个按钮，其中【**导入文件】**可以选择文件导入【代码编辑区】；

- 点击**【开始编译】**，将会编译【代码编辑区】中的代码，并在右侧【目标代码区】显示Pcode结果，下方【运行信息区】会显示编译成功或者显示错误信息，；

- 点击【**解释运行】**前需要先进行正确编译，编译通过后，如果程序中有输入数据，先在【输入框】中输入数据（数据可多不可少），再点击解释按钮，否则会弹出输入变量不够，左下的【输出框】会显示解释结果；

最后通过【**保存目标代码**】可以保存目标代码至计算机的指定位置，

整体情况如图6所示：

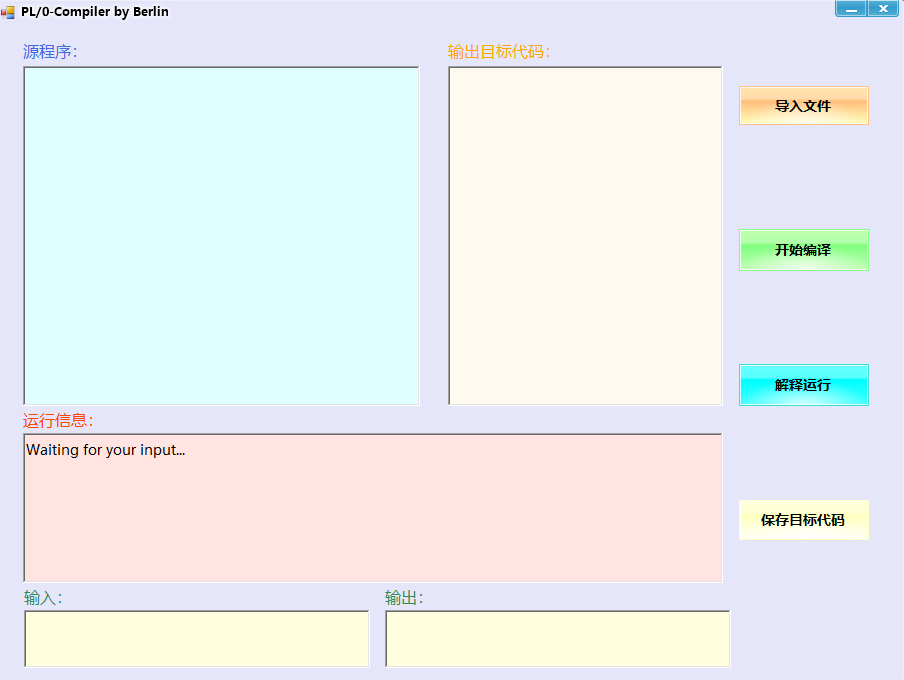


图6 初始界面

导入测试样例2后进行编译的结果：

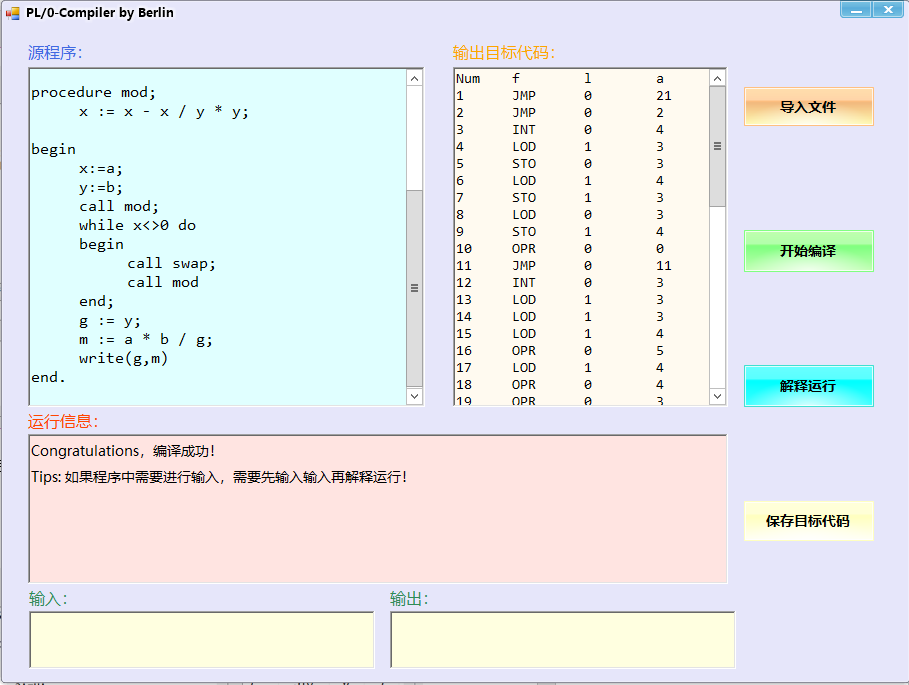


图7 成功编译

点击【解释运行】后的结果，可以看见右下角【输出栏】正确显示了a，b的最大公约数9以及最小公倍数135：



图8 成功运行

编译存在错误的测试样例3后的结果如下图所示，可以发现尽管从第2行开始就开始出现编译错误，但是没有影响编译的进行，一直检查到了最后发现了缺失’.’的错误，并且也输出了“目标代码”：

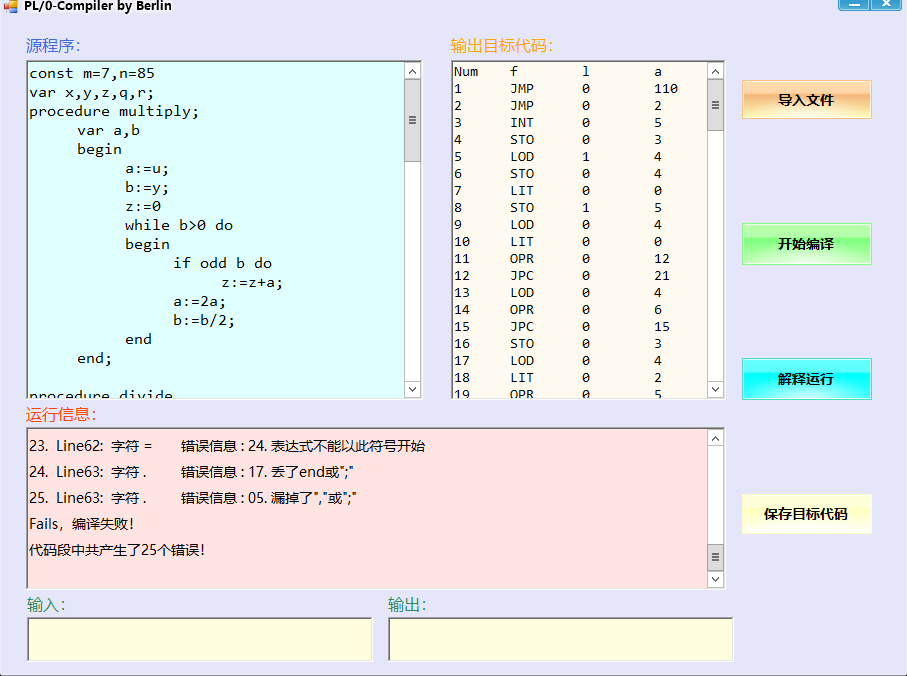


图9编译错误

## 

图10 导入源文件页面

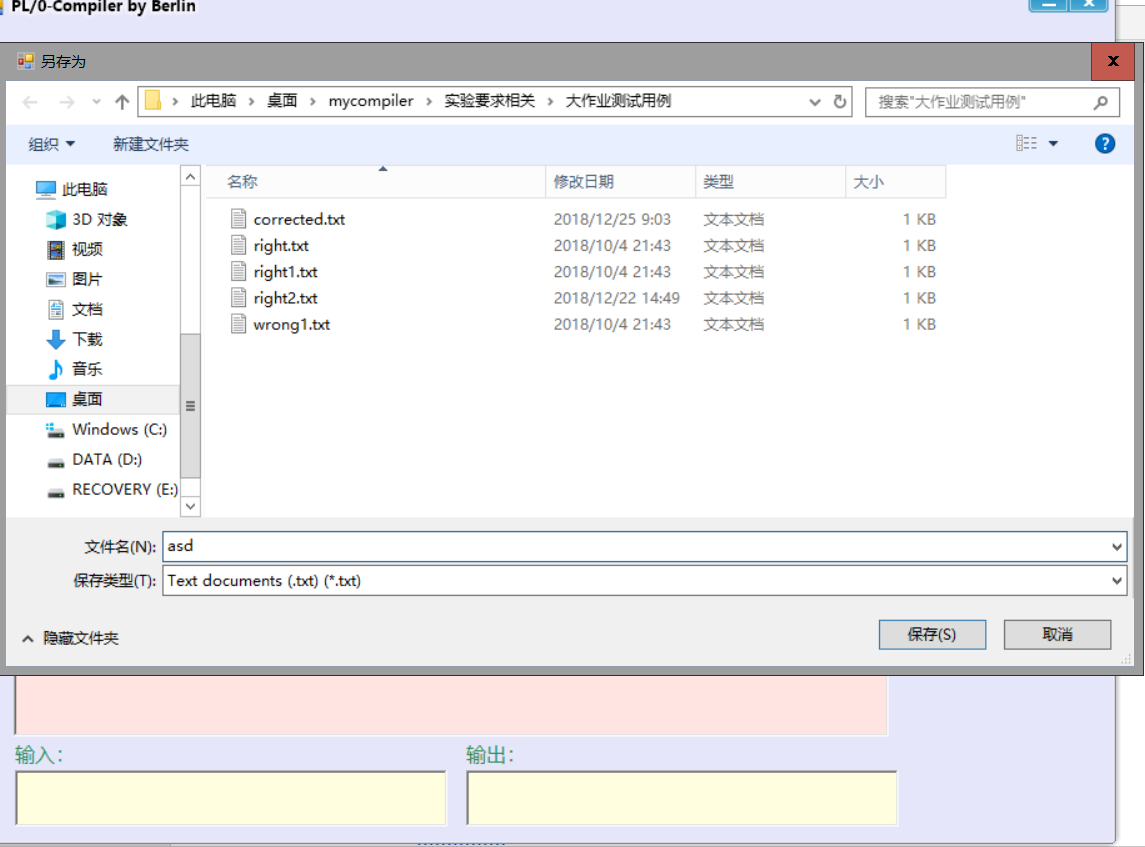


图11 导出目标代码页面

## 五、实验感想

通过此次实验对于PL/0编译系统的编写，让我在这个过程中了解了整个编译系统的“生命流程”，纸上学来终觉浅，绝知此事要躬行。课堂上学习的知识较为零散、而真正在编写的时候才真正体会到了之前这些零散的知识如何组建在一起，最终成为一个较为完整的编译系统。

PL/0文法在教材上有一定的概念介绍，包括语法图以及依赖图等等，但是刚开始看完之后还是一头雾水，词法分析模块由于之前已经编写完成，因此此次可以继续复用，但是之前的词法分析器其实是在主界面调用的，并且只需要输出分析结果，当时只是构建了索引与token类型的数组映射，但是真正的词法分析器是需要在语法分析器中调用的，如果依然采用这种索引的方式，就会给代码的编写带来很多不便，因此对之前的词法分析器又进行了部分的重写，同时相比教材上规定的getsym，我实现的词法分析器也略有不同，是对源程序所有单词直接预读进入到单词表中，而在语法分析器中只需要调用Getsym就能实现相同的取下一个单词的效果。而语法分析模块是最为复杂的部分，因为它连接着符号表以及目标代码的生成，其中大部分的错误处理也体现在语法分析部分，符号表部分之前一直都云里雾里地，特别是关于地址反填技术时与Pcode指令之间的相关联系，在语法分析过程中实现Pcode集指令的对应需要之前对Pcode 10个操作码，总计24种指令的含义进行了解，才能在语法分析各个模块中合理生成Pcode指令；其次对于PL/0的头符号和后继符号集其实很有规律可循，可以通过定义三个头符号集合备用，以后对于每个语句块的FIRST集以及FOLLOW集的编写可以复用上一个语句块的FOLLOW集，在其基础上进行删改；而对于错误处理的编写，之前也不知道从何将它们覆盖，后来在实际的语法分析时，会发现都是水到渠成的过程。

在程序编写的过程中，让我也意识到，在编写较为复杂的程序时，变量的规范命名以及单步调试使用的重要性，之前在进行运行语法分析程序时，经常陷入死循环，只有通过单步调试才能发现循环点，从而进行修正，而规范的变量命名也有助于在debug的时候降低调试的难度。

尽管在设计过程中遇到了一些问题，但经过一次又一次的思考，一遍又一遍的检查终于找出了原因所在，也暴露出了之前我在之前的学习中仍然存在欠缺和经验不足，课下通过和同学之间的交流，让我对于编译系统的构成有了更加深入的理解，尽管设计还是存在一些遗憾（比如输入部分用户操作体验还可以继续提升；文法本身还可以进一步进行加强等等），这些都是还可以进一步提升的方面。

通过本次实验对于PL/0编译系统的编写，使我对于编译系统的工作流程有了更加清晰的认识，而且在程序代码的编写以及可视化界面的设计上也是对上个学期C#知识的回顾与应用；实践出真知，通过这次课程设计使我懂得了理论与实际相结合是很重要的，只有理论知识是远远不够的，只有把所学的理论知识与实践相结合起来，从理论中得出结论，才能提高自己的实际动手能力和独立思考的能力，通过这一次的亲身实践，使我掌握的知识不再是纸上谈兵，受益匪浅。

## 六、测试报告

**测试程序及测试结果**

**测试用例1**

教材P309例14.1 求解鸡兔同笼问题（对错误进行了修正）：

const z=0;

var head,foot,cock,rabbit,n;

begin

n:=z;

read(head,foot);

cock:=0;

while cock<=head do

begin

rabbit:=head-cock;

if cock\*2+rabbit\*4 = foot then

begin

write(cock,rabbit);

n:=n+1

end;

cock:=cock+1

end;

if n=0 then write(0,0)

end.

Input：

3 5

5 12

Output：

0 0

4 1

**测试用例2**

教材P310例14.2 求最大公约数和最小公倍数

const a = 45, b = 27;

var x,y,g,m;

procedure swap;

var temp;

begin

temp := x;

x := y;

y := temp

end;

procedure mod;

x := x - x / y \* y;

begin

x:=a;

y:=b;

call mod;

while x<>0 do

begin

call swap;

call mod

end;

g := y;

m := a \* b / g;

write(g,m)

end.

Output：

9 135

**测试用例3**

教材P320例14.7.2 语法错误样例

const m=7,n=85

var x,y,z,q,r;

procedure multiply;

var a,b

begin

a:=u;

b:=y;

z:=0

while b>0 do

begin

if odd b do

z:=z+a;

a:=2a;

b:=b/2;

end

end;

procedure divide

var w;

const two=2,three:=3;

begin

r = x;

q:=0;

w:=y;

while w<=r do

w:=two\*w;

while w>y

begin

q:=(2\*q;w:=w/2);

if w<=r then

begin

r:=r-w

q:=q+1

end

end

end;

procedure gcd;

var f,g;

begin

f:=x;

g:=y

while f<>g do

begin

if f<g then g:=g-f;

if g<f then g:=f-g;

z:=f

end;

begin

x:=m;

y:=n;

call multiply;

x:=25;

y:=3;

call divide;

x:=84;

y:=36;

call gcd;

call x;

x := gcd;

gcd = x

end.

Output：

Fails，编译失败！

代码段中共产生了26个错误！

**测试样例4**

const d = 10;

var a,b,c;

procedure test1;

const a = 2,b = 3;

procedure test2;

var a,b,c;

begin

read(a,b);

write(a,b)

end;

begin

write(a,b)

end;

call test1.

Output:

2 3

**测试样例5**

var x,y,m,n,pf;

const true=1,false=0;

procedure prime;

var i,f;

procedure mod;

begin

x:=x-x/z\*y;

end;

begin

f:=true;

i:=3;

while i<m do

begin

x:=m;

y:=i;

call mod;

if x=0 then f:=false;

i:=i+2;

end;

;

if f=true then

begin

write(m=1);

pf:=true;

end

end;

begin

pf:=false;

read(n);

n:=3;

while n >= 2 do

begin

write(2);

if n=2 then pf:=true;

m:=3;

while m<=n do

begin

call prime;

m:=m+2;

end;

read(n);

end;

if pf=false then write(0)

end

end

Output:

Fails，编译失败！

代码段中共产生了5个错误！

**测试样例6**

const a=21,flag=4;

var b,c,d;

begin

b:=7;

write(a+b);

repeat

b:=b-1;

if b>flag then write(b);

else write(flag);

until b<0;

end.

Output：

Fails，编译失败！

代码段中共产生了1个错误！（Line9: 字符 else 错误信息 : 19. 语句后的符号不正确）

## 七、参考资料

[1]张莉等：《编译原理及编译程序构造》，清华大学出版社，2011年6月。

[2]陈英等：《编译原理》（第二版），北京理工大学出版社，2006年2月。